

RICE UNIVERSITY

**Semi-Automatic Synthesis of Parameterized
Performance Models for Scientific Programs**

by

Gabriel Marin

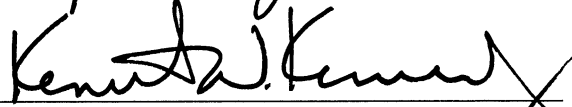
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

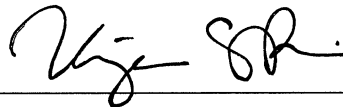
APPROVED, THESIS COMMITTEE:



John Mellor-Crummey
Senior Faculty Fellow, Computer Science



Kenneth W. Kennedy
Ann and John Doerr University Professor,
Computational Engineering



Vijay S. Pai
Assistant Professor, Electrical and
Computer Engineering

Houston, Texas

April, 2003

UMI Number: 1414280

UMI[®]

UMI Microform 1414280

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Semi-Automatic Synthesis of Parameterized Performance Models for Scientific Programs

Gabriel Marin

Abstract

Building parameterized performance models of applications in an automatic way is difficult because of the large number of variables that affect performance, including architecture-dependent factors, algorithmic choices and input data parameters. In general, application performance is a non-convex and non-smooth function in this multivariate parameter space.

This thesis describes techniques to measure and model application characteristics independent of the target architecture. This approach produces an architecture-neutral model for an application. For predictable applications, such models have a convex and differentiable profile. Our approach succeeds in modeling the most important application factors that affect performance and enables us to explore the interactions between a target architecture and application characteristics.

To date, work has concentrated on modeling the performance of intervals of sequential computation. Our models are designed to characterize node performance between synchronization points in parallel programs, with the eventual goal of modeling the performance of parallel applications.

Acknowledgments

First of all, I would like to thank my advisor, John Mellor-Crummey, for his support and continuous guidance. His advice was a substantial help in my understanding of performance analysis. Also, John's careful reading of the draft of my thesis greatly improved the clarity of the final version.

Next, I would like to thank Ken Kennedy and Vijay Pai for their contribution as members of the committee of this thesis. I am grateful to Richard Hanson for the discussions we had regarding modeling strategies.

My parents have always encouraged me and guided me to independence. I am grateful to them for their generosity.

Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vi
List of Tables	viii
1 Introduction	1
2 Background	4
2.1 Performance Prediction Techniques	5
2.1.1 Profile-based approaches	5
2.1.2 Simulation-based approaches	7
2.1.3 Pencil-and-paper analytical methods	8
2.2 Memory Performance Analysis	10
2.2.1 Understanding How Caches Work	10
2.2.2 General Techniques for Memory Performance Analysis	14
3 Toolkit Design and Implementation	17
3.1 Updates to the EEL library	18
3.2 Static Analysis	21
3.3 Dynamic Analysis	21
3.3.1 Collecting Execution Frequency Histograms	22
3.3.2 Collecting Communication Data	28
3.3.3 Monitoring Memory Access Behavior	31
3.3.4 General Features of the Binary Instrumenter	33

3.4	From Data to Parameterized Models	35
3.4.1	Building models of the execution frequency	35
3.4.2	Building models of the memory access pattern	37
3.5	Mapping the Models to a Target Architecture	43
4	Experiments	46
4.1	Characterizing the PSTSWM Application	46
4.1.1	Constructing FLOP Count Models for PSTSWM	46
4.1.2	Validating the FLOP Count Models	51
4.1.3	Memory Reuse Distance Models for PSTSWM	53
4.2	FLOP Count Models for CRM	60
5	Conclusions	64
	Bibliography	66

Illustrations

2.1	A loop nest to multiply a pair of matrices.	13
3.1	Toolkit architecture overview.	17
3.2	Step by step example for inserting counters into a CFG	28
3.3	Example of reuse distance histogram. All references with reuse distance less than the cache size are hits.	38
3.4	The error distribution of the average memory reuse distance model for Sweep3D. The x axis represents the problem size and the y axis represents the fraction of accesses. The model has less than 1% error for three quarters of the accesses and the tendency is to have a higher precision for larger problem sizes.	39
3.5	The surface on the right is the parameterized model for one of the most frequently executed memory accesses in Sweep3D. On the left is the original data collected for that instruction. The x axis represents the problem size, on the y axis is normalized number of accesses and the z axis represents the reuse distance.	41
3.6	Example of predicting the cache misses using a memory reuse distance model	42
3.7	The collected data and the constructed model for another frequently executed memory access from Sweep3D. The x axis represents the problem size, on the y axis is normalized number of accesses and the z axis represents the reuse distance.	43

4.1	The number of FpAdd operations measured for all valid NLON values less than 256	48
4.2	The number of FpAdd and FpMult operations, measured for $NLON = 2^k$, $k = \overline{4,9}$	49
4.3	The accuracy of the constructed model in two variables, relative to the measured data collected by our tool.	51
4.4	L1 miss counts predicted vs. measured for the four most important computational routines in PSTSWM.	55
4.5	Dynamic memory operation count predicted vs. measured.	55
4.6	The innermost loop of routine <i>rs piv</i>	56
4.7	Depending on the alignment of each column of the SUM array, the two elements referenced in one iteration can be located either in the same memory block(left), or in different memory blocks(right).	58
4.8	Data collected for one of the accesses to the SUM array. About half of the reuse distance histograms have a similar fraction of accesses with distance zero (10%). The other histograms, marked with an arrow, seem to have a variable fraction of accesses with distance zero.	59
4.9	L1 miss count predicted vs. measured after the divergent data points have been removed.	59
4.10	Dynamic memory operation count predicted vs. measured for the binary compiled with <code>-dalign</code>	60

Tables

3.1	Classification of SPARC native instructions into generic RISC classes.	45
4.1	Two variable routine-level models of the count of dynamic floating point instructions and their relative error (first part).	62
4.2	Two variable routine-level models of the count of dynamic floating point instructions and their relative error (second part).	63

Chapter 1

Introduction

Characterizing and modeling the performance of parallel applications has been a long-standing goal of computer science research. An accurate performance model for an application has many uses including understanding its scalability, providing guidance for resource selection at launch time, guiding on-line performance monitoring and run-time adaptation, and providing input for design of future architectures that will meet an application's resource needs.

Building accurate performance models for parallel applications is difficult. Simply knowing the number of floating-point operations a scientific application executes provides little indication of its performance. Scientific codes rarely achieve peak performance. On a single node, memory hierarchy latency and bandwidth are significant limiting factors. Also, an application's instruction mix can dramatically affect performance; today's superscalar processors can execute multiple instructions in parallel if they are provided with the right mix of instructions. For parallel programs, communication frequency, communication bandwidth and serialization complicate the situation further.

Traditionally, scientists have manually built analytical models of an application's performance. This requires a thorough understanding of the algorithms that are used in an application, as well as of their implementation. The advantage of manually constructing such models is that they can be very precise. However, building accurate models in this way is laborious and painstaking.

The thesis of this work is that it is possible to characterize the performance of sequential and parallel applications in a semi-automatic way, with a reasonable accu-

racy. Building parameterized performance models of parallel and single node applications is difficult because of the large number of variables that affect performance, including architecture-dependent factors, algorithm/application choices and input data parameters. Moreover, these factors interact in complex ways, producing a performance function that is non-convex and non-smooth in this multivariate parameter space.

This thesis proposes separating the contribution of application-specific factors from the contribution of characteristics of the target architecture. The benefits of this approach are two fold: first, by modeling the application-specific factors we can build architecture-neutral models, portable across different platforms; second, models that describe the algorithmic and application choices are most of the time monotonic polynomial functions that are easier to synthesize. Our approach succeeds at building parameterized analytical models of the most important characteristics of black-box applications in a semi-automatic way.

We build models using information from both static and dynamic analysis of an application's binary. By working with application binaries instead of program source code, we are able to build language-independent tools that can naturally analyze applications with modules written in different languages or linked with third party libraries. By analyzing binaries, the tool can also be useful both to application writers and to compiler developers by enabling them to evaluate performance and scalability of algorithms as well as to verify the effectiveness of their manual or compiler-based optimizations. We use static analysis to construct the control flow graph of each routine in an application and to examine the instruction mix inside the most frequently executed loops. We use dynamic analysis to collect data about the execution frequency of basic blocks, to collect information about synchronization among processes and to measure reuse distance seen by individual memory accesses.

The rest of this thesis is organized as follows. Chapter 2 presents background information and recent related work. Chapter 3 describes in more detail the goals, the

design and the implementation of our performance toolkit. Chapter 4 discusses experiments using our toolkit to synthesize performance models for several applications. Chapter 5 presents our conclusions and identifies directions for future work.

Chapter 2

Background

Performance analysis and modeling comprise the process of building mathematical constructs to describe performance characteristics of a computer system. Performance prediction represents the computation of an estimated execution time of a program on a given architecture. Predicting the execution time of an application under varying conditions is one of the most important, and yet most difficult, aims of performance analysis research.

Performance analysis research is useful to several groups of people. Compiler writers, application developers and computer architects have the most interest in this research area. Application developers need performance analysis to understand the causes of inefficiency in their programs, guide procurements and guide the selection of an appropriate set of resources at application launch time. Compiler writers may use performance tools to understand opportunities for optimization, estimate potential payoff of optimizations on today's architecture or to evaluate the performance of the code they generate for the architectures of tomorrow that are in the design stage. Finally, computer architects may use performance models to understand the causes of performance inefficiencies, estimate their impact, and explore the impact of architectural changes.

Existing performance prediction methods range from back-of-the-envelope estimates to detailed analytical models or cycle-accurate simulations. Recent work in this field focused on finding an approach that will cut the time overhead required by the full-scale simulators while maintaining a reasonable level of accuracy. In the next section, I present some of the recent work in this field. In section 2.2, I describe the

existing techniques for analyzing memory access behavior.

2.1 Performance Prediction Techniques

We can divide the existing techniques into three main categories: profile-based approaches, simulation-based approaches and pencil-and-paper analytical methods.

2.1.1 Profile-based approaches

Profile-based performance prediction methods use hardware performance counters or code instrumentation to collect performance data during an application's execution. Afterwards, a post-processing tool analyzes this data to determine places where most time is wasted and, hence, are the most profitable to be optimized, or to compute an estimate for the application's execution time.

Hardware performance counters are a set of special registers available on most modern microprocessors. These registers count events related to events that take place inside the processor or the memory subsystem when application software is executed. Hardware performance counters can capture statistics about executed instructions or memory hierarchy behavior with a minimal time overhead. Hardware performance counters are especially useful in gathering data about the interaction between the application and the hardware it is running on. The method presented in this thesis constructs models for characteristics of an application that are not dependent on the hardware; therefore, hardware counters are not used by our toolkit.

Alternatively, code instrumentation can be used to collect information about applications' performance. Code instrumentation can be performed either on source code or on object code. Object code instrumentation can be classified further as dynamic instrumentation, link-time instrumentation, or static instrumentation. In recent years, several tools/libraries for binary instrumentation have emerged. DynInst [9] is a portable application program interface (API) that enables development of tools and applications that require run-time code patching. Dynamo [5] is a run-time dy-

dynamic optimization system that focuses on optimization opportunities which manifest themselves only at runtime and, hence, are difficult for a static compiler to exploit. Purify [15] is a well known tool for memory error and leak detection that uses object code instrumentation at link-time to monitor memory allocation, deallocation and memory accesses performed by an application. The category of static binary instrumentation is represented by EEL [18], ATOM [28] and Etch [26] libraries that enable tools and applications to analyze and modify binary programs without being concerned with low-level architecture details.

Snively, Wolter and Carrington [27] assume that a parallel application’s performance is based on two major factors: its single node performance and its use of a communication network. They consider that the network interconnect contribution can be estimated with reasonable results by a network simulator. For single processor performance, their investigations focus on memory-bound codes, such as the NAS Parallel Benchmarks [4] (NPB) kernels. Therefore, they use the “rule of thumb” that the per-processor performance of an application is predominantly a function of how it exercises the memory subsystem. Their performance prediction method consists of collecting Machine Signatures, characterizations of the rates at which a machine can execute fundamental operations independent of the particular application, and Application Profiles, summaries of the fundamental operations to be carried out by the application in abstraction of any particular machine. Then the application profiles are mapped onto the machine signatures using a Convolution Method. For the NPB kernels they considered, the errors of the estimates are less than 20% in all cases. The cost of the modeling is a sixty-fold slowdown for collecting the Application Profiles and a roughly equal amount of time spent in gathering the Machine Signatures. However, the latter information can be used for other applications without any further overhead. This time overhead is favorable when compared to a cycle-accurate simulation that can have as high as a six orders of magnitude slowdown without a significant gain in accuracy for these memory-bound kernels.

However, considering the requirements of executing profiled applications at full scale to get node characterizations, and simulating communication traces at full scale to estimate the synchronization cost, we consider that even a sixty-fold slowdown is an important deterrent for using this method on large data sizes for which the original program can run for days. The memory-bound rule can be safely applied to many scientific applications, but this should not be a general assumption. Other factors, such as the instruction schedule dependencies, can affect the estimates more than just a few percent. Snaveley et al’s work is the closest in concept to the method presented in this thesis. We also focus on collecting application characteristics independently of the target machine. Unlike Snaveley et al., we consider not only models for memory-bound applications; our models consider a variety of factors that can affect performance, including memory hierarchy latency, the instruction schedule dependencies and instruction mix inside the most executed loops, and the synchronization among processes. More importantly, we build scalable models of these application characteristics. As a result, we can predict performance of an application for larger problem sizes than the ones that are practical for monitoring and simulating at scale.

2.1.2 Simulation-based approaches

Performance prediction methods based on simulation consist of executing an application in conjunction with a program that emulates the target architecture. Since each dynamic instruction must be simulated, these methods have a significant time overhead. On the other hand, a detailed simulation can produce a very accurate prediction of the application’s performance. Trace-based simulators are a common style tool supporting this type of method. To use a trace-based simulator, the application is instrumented using a profile-based tool to collect a trace of fundamental events that occur during its execution. The resulting trace file is then fed into a program which simulates only a fraction of the dynamic instructions originally executed by the application. Hence, this method is much faster than a cycle-accurate simulation.

However, speed comes at a cost and the trade-off in this case is the accuracy of the prediction and the level of detail about the phenomena taking place inside the simulated machine. A general drawback of simulators is that the user has to simulate the application under study for each data set of interest. Still, simulation or trace-based simulation can be used together with other techniques to construct scalable models of an application's performance.

The POEMS [2] project by Adve et al., is an environment for end-to-end performance modeling of complex parallel and distributed systems. The POEMS modeling effort spans the domains of application software, runtime libraries, operating system, and hardware architecture. The POEMS effort aims to assemble component models from these different domains into an end-to-end system model. The composition process is specified using a generalized graph model of a parallel system, combined with interface specifications describing the component behaviors and evaluation methods. The POEMS framework is a complex system that aims at building complete end-to-end models using a large range of modeling paradigms including analysis, simulation and direct measurement. The drawbacks of this approach are the high complexity of the system with possible long execution times for the simulators and the system's dependency on the task graph that can be generated only by the Rice *dHPF* compiler [22]. This thesis explores another approach for building scalable models of an application's computation using dynamic analysis of application binaries without any special compiler support.

2.1.3 Pencil-and-paper analytical methods

The pencil-and-paper method is the traditional technique for building performance models. It requires a deep understanding of the algorithms that are used in an application, details about their implementation and a good knowledge of the synchronization techniques chosen by the developers. As a result, this method is not widely used in industry. It is used only by highly skilled researchers. However, it will continue to be

used until more automated methods become mature enough to achieve the desired levels of accuracy. On a smaller scale, pencil-and-paper models can also be used to validate models constructed with other techniques.

Sundaram-Stukel and Vernon [29] analyze and construct a LogGP performance model for Sweep3D, a wavefront application with a complex synchronization structure. They focus on building accurate models of MPI communication primitives using micro-benchmarks on two or four nodes. They show that the LogGP model predicts with high accuracy the measured application execution time for large problem sizes running on 128 nodes or more. Stukel and Vernon found poor scaling beyond two thousand nodes for the Sweep3D application using the current algorithm, due to synchronization overhead. We also plan to use the LogP [11, 13] or LogGP [3] models to characterize the time spent in communication and synchronization primitives. The LogP models combined with our single-node modeling approach can provide the support to construct models of entire parallel applications.

Hoisie et al. [16] studied the performance of wavefront algorithms implemented using message passing on 2-dimensional logical processor arrays. Wavefront algorithms are widely used in parallel computing since they enable parallelism in computations that contain recurrences. Hoisie’s group is focused on predicting performance of wavefront applications on cost-effective machines that have non-uniform network topologies, such as in a cluster of SMPs interconnected by a network of lower dimensionality. This type of architecture is the main candidate for building the multi-TeraOp systems of tomorrow. In their work on wavefront applications, they successfully captured the communication pattern in their model resulting in an accurate performance model validated on a cluster of Origin 2000 machines with up to 1024 processors.

While the pencil-and-paper approach has produced some of the most accurate models to date, the obvious disadvantage is the human effort required to construct them.

2.2 Memory Performance Analysis

With the continuously widening gap between the peak performance of microprocessors and the available memory bandwidth, it is unanimously accepted that memory hierarchy response is the factor most limiting node performance. Cache memories are small, fast buffers placed between the processor and the main memory to help hide the large latency of memory accesses. Caches are effective only if an application exhibits temporal or spatial cache reuse. For an application with a working set much larger than the cache size, a stream of random accesses to the working set will have little or no cache reuse. To increase the reuse in the cache, one can either increase the size of the cache to be approximately the size of the working set or one can optimize the application to exploit temporal reuse of data that is already in the cache before evicting it. In this section I describe the main techniques for assessing how an application uses the memory hierarchy. First, I review how caches work.

2.2.1 Understanding How Caches Work

Caches are characterized by three principal parameters: cache size, block size, and associativity. The cache size is the total capacity of the cache in bytes. The cache is divided into a number of equally sized blocks called cache lines. All operations with the next level of the memory hierarchy are performed with a cache line granularity. The size of the cache line defines the number of bytes that are fetched from memory when a cache miss occurs, or the number of bytes written to memory when a modified cache line has to be evicted. The capacity of the cache (C) is equal to the number of blocks (N) times the block size (B).

The associativity represents the number of unique cache lines in which a memory block can reside. If a memory block can be loaded into one cache line only, the cache has a direct mapping. If the block can be loaded into any cache line, the cache is fully associative. If the block can reside in a set of exactly k cache lines, then the cache is k -way set-associative.

The level of associativity built into a cache memory affects the performance of the entire system. Fully associative caches are expensive and difficult to build; therefore, caches used in practice have a much lower level of associativity. To understand how the associativity level can affect the performance, one must consider how memory blocks are mapped into the cache lines depending on the type of cache. Let R be the ratio between the size of the memory and the capacity of the cache. Because the capacity of the cache is much smaller than the size of the memory, on average R blocks from the memory compete for each one of the N blocks in the cache. In a fully associative cache any of the $R * N$ memory blocks can occupy any of the N cache blocks. In a k -way set-associative cache, $R * k$ memory blocks compete for a set of k blocks in the cache. Even if the ratios are equal, there is a difference in the size of the two sets. For a fully associative cache, the layout of the data in memory has little importance. For a system with a lower level of associativity (k), if the accesses to memory are referencing only one set of $R * k$ memory blocks, then only the corresponding set of k cache lines is used. Therefore, more optimizations are necessary when compiling an application for a machine with a low level of cache associativity, to ensure that the most frequently accessed data structures are uniformly distributed over the N/k sets.

A cache hit is classified as a temporal reuse or a spatial reuse. We say that a memory access has temporal reuse in the cache if a previous access to the same location fetched the memory block into the cache or kept it from being replaced. A memory access has spatial reuse in the cache if a previous access to a location from the same memory block caused the cache line to be in the cache. A cache configuration with a cache line of size one has no spatial reuse.

Cache misses can be classified in three categories: compulsory misses, capacity misses and conflict misses. Compulsory misses are also called cold misses because they are caused by a cold cache. A cold miss is produced by accessing a memory block that was not referenced before. If an application has to access each memory block only once in the entire execution, then all memory accesses will result in compulsory

misses. Reducing the number of compulsory misses requires explicitly prefetching the data. Another (theoretical) solution is to reduce the number of blocks in memory by increasing the block size.

An access to a memory block b_i is a capacity miss if and only if at least N different other blocks were referenced since the last access to b_i . This is equivalent to saying that capacity misses are misses that would occur in a fully associative cache with an LRU replacement policy and that are not compulsory misses. The number of capacity misses can be reduced by either increasing the number of blocks in the cache or by restructuring the application to reuse blocks while they are in cache, if there is potential temporal reuse that is not being exploited.

A reference that hits in a fully associative cache and misses in a k -way set-associative one is called a conflict miss. It means the referenced block b_i was accessed in the recent past because the reference is a hit in a fully associative cache, but at least k other different memory blocks from the set of $R * k$ blocks that compete for this set of cache lines were also referenced since then, causing the eviction of block b_i . Conflict misses are the most difficult to model or to predict since they are the result of a complex interaction between the characteristics of the cache sub-system, the layout of the data in memory and the access pattern used by application to access this data.

In fact, in certain cases an application can have a negative number of conflict misses, meaning the program will see more misses with a fully associative cache than with a 2-way set-associative. This behavior can be illustrated with an unoptimized matrix-multiply program. Figure 2.1 presents the “C” code for a simple matrix-multiply program. Arrays X, Y and Z are laid out in memory in row-major order - consecutive elements of a row occupy consecutive locations in memory. If we consider a case where the matrix line size expressed in bytes (M') is less than the size of the cache (C) but the number of elements in a line or column (M) is greater than the number of blocks in the cache, we would expect to see only spatial reuse for

consecutive stride one accesses to X performed in the inner loop and no temporal reuse from the middle loop in case of a fully associative cache.

```

for( i=0 ; i<M ; i++ )
  for( j=0 ; j<M ; j++ )
  {
    Z[i][j] = 0;
    for( k=0 ; k<M ; k++ )
      Z[i][j] += X[i][k]*Y[k][j];
  }

```

Figure 2.1 : A loop nest to multiply a pair of matrices.

The program references a full row of matrix X and a full column of matrix Y before reusing the same row of X in the next iteration of the middle loop. We can compute how many blocks are accessed by the inner loop. A line of the matrix X occupies M'/B blocks, where B is the block size in bytes. An entire column of Y occupies M blocks, where M is the number of elements in a column, because two consecutive elements of a column occupy two different memory blocks. From the hypothesis, M is greater than the number of blocks in the cache. As a result, one column of Y occupies more blocks than the total number of blocks in the cache. The program cannot see temporal reuse for the accesses to A from the middle loop.

For a 2-way set-associative cache, accesses to array Y will generate a lot of conflict misses, many times causing the eviction of an another element of Y, referenced not long before. This effect will permit some of the elements of X to stay in cache for the entire execution of the inner loop, and the application will experience temporal reuse in the next iteration of the middle loop. Accesses to the Y array produce conflict misses but because they are already counted as capacity misses and X experiences more hits than in the case of a fully associative cache, by the definition above, accesses to X have a negative number of conflict misses.

2.2.2 General Techniques for Memory Performance Analysis

Existing techniques for understanding how an application uses the memory hierarchy include cache simulator methods, compile time analysis techniques, profile-based methods and dynamic code monitoring and optimization.

Cycle accurate simulators were described in section 2.1.2. They emulate the entire target machine, and therefore, they must simulate the cache in the process. Cache simulators do not need to simulate execution of all of an application's instructions, only the memory references. The simulation can be performed either offline, on a trace of references collected with the help of a profiling tool, or on-the-fly by executing an instrumented version of the program. Except for compile time analysis methods, all the other methods may make use of a cache simulator. The disadvantages of a trace-based cache simulator are the possibly large disk space needed to store the traces with tens or hundreds of millions of memory references, and the need to separately collect and simulate a trace of memory references for each configuration of input parameters and cache parameters that must be analyzed.

Recent work by Mueller et al. [25] proposes the use of regular section descriptors (RSD) and partial address traces to represent the data traces in constant space, solving the first disadvantage presented above. Mueller et al. use dynamic binary rewriting to collect the partial data traces. However, the presented algorithm deals only with regular access patterns of streams that have a constant stride. Extending the algorithm to handle arbitrary strides might be possible, but with a big cost increase.

This thesis presents another method for analyzing the data access pattern of an application by characterizing the memory reuse distance for each memory reference. Memory reuse distance is a measure of the number of distinct memory locations that are accessed between two references at the same datum. Comparing the reuse distance information seen by a memory reference and the number of blocks present in a cache sub-system (see Section 2.2.1), provides a direct indication if the memory access is a

hit or not in the considered cache configuration. Although we collect detailed enough information to predict the miss ratio independently for each memory reference in the program, the output is rather compact and requires an almost constant space. Also, we can deal with strides that are arbitrary polynomial functions, not only constants. Furthermore, this work presents a method to predict how the memory reuse data extrapolates at a different problem size that we did not collect data for.

The MHSim [23] memory simulator by Mellor-Crummey, Fowler and Whalley, uses source-to-source translation of Fortran programs to instrument all memory accesses with code that simulates a parameterized cache system on-the-fly. Next, the MHSim simulator processes the annotated data trace and correlates cache miss information to line numbers in the source code. However, since MHSim uses source-level instrumentation, the collected trace of memory references may not correspond to the actual order in which accesses are executed by an aggressively optimized binary. For instance, source-level instrumentation might prevent loop transformations such as tiling, that change the order in which data is accessed.

Lebeck and Wood [19] implemented CProf, another cache profiler. They use static instrumentation of application binaries to substitute the memory references with calls to a function that simulates caches online. This thesis uses a similar mechanism for static binary instrumentation, but we process the stream of accesses online to create a compact representation of the data access pattern exhibited by the application instead of simulating the cache on-the-fly.

Ding and Zhong [12] collect histograms of the memory reuse distance of all memory references in an application to provide a measure for program locality. They use a modified compiler front-end to insert instrumentation code that assigns the information collected to data structures in the source code. The memory performance analysis in this thesis is also based on collecting reuse distance information. Though the similarities stop here, Ding and Zhong’s study is the closest to our work on memory locality analysis. Since we are interested to predict the performance of

applications executed on arbitrary architectures, and we consider detailed information such as instruction schedule dependencies inside the most frequently executed loops, we must recover the access pattern seen by individual references in order to decide if a longer latency corresponding to a specific miss can be hidden by other useful work. As a result, we characterize each memory reference individually, unlike Ding and Zhong who produce a source-level centric view of how the program accesses each data structure.

The category of compile time techniques is represented by the work of Ghosh, Martonosi and Malik [14]. They describe methods for generating and solving cache miss equations to get a detailed representation of cache misses in loop-oriented scientific codes. Ghosh et al.'s approach uses static analysis of the source code to generate a set of linear Diophantine equations whose solutions correspond to potential cache misses. This approach is suitable for compilers to guide memory optimizations.

The next Chapter presents the design and implementation choices for our performance analysis toolkit.

Chapter 3

Toolkit Design and Implementation

Our performance modeling toolkit is a collection of scripts and programs designed to facilitate performance analysis and the construction of scalable performance models for scientific applications. Figure 3.1 presents a high level view of the components of our toolkit.

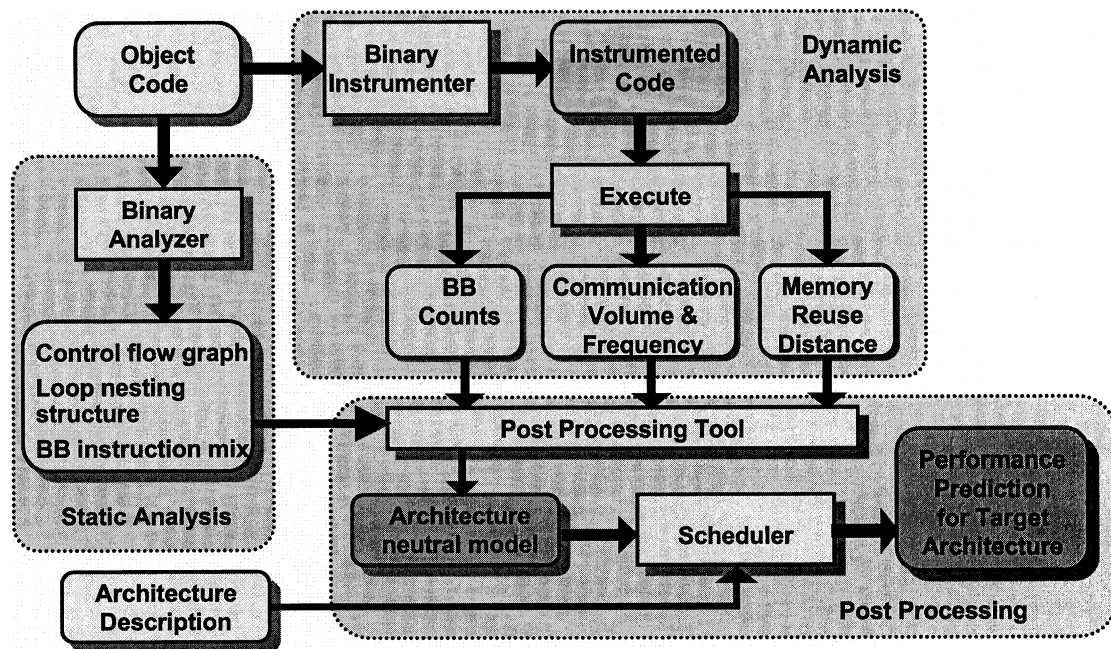


Figure 3.1 : Toolkit architecture overview.

We use static and dynamic analysis of binaries to derive architecture-neutral models for an application. We aim to predict the performance on any given target architecture by convolving the models with a description of the resources available on the target machine.

Analyzing binaries instead of source code has several advantages: we do not need separate handling for each source language, we do not need to analyze a possible large number of source files located in multiple different directories, and we can deal with situations when the source code is unavailable such as when the application is linked against libraries available only in binary form. Other issues avoided are the usual discrepancies between the structure of source code and the structure of highly-optimized object code. Also, it is easier to analyze a mixture of low-level instructions that a machine can execute directly, with a predictable latency, than to try estimating the execution cost for high-level language constructs. Finally, instrumentation of object code, as opposed to source code, can properly account for the effects of aggressive optimizations; source level instrumentation may inhibit key optimizations that are critical for high performance.

3.1 Updates to the EEL library

Our binary analysis tools are built on top of the Executable Editing Library (EEL) [18]. The EEL library was released in 1995 and it has not been publicly maintained since then. Since both the Sun Forte compiler and the GNU gcc compiler were updated constantly in this time, we had to bring EEL up to date on understanding control flow mechanisms used by today's compilers.

Special effort was put into recovering the control flow of indirect jumps. We greatly extended the previous method for correctly understanding the dispatched indirect jump instructions which account for the vast majority (often 100%) of indirect jumps in a program. Also, we replaced the old method for dealing with register indirect jumps which considered the target address computed along one path only. The new improved method is a fix point algorithm that checks each new path leading to the jump instruction and adjusts the set of possible targets until no new path is discovered.

Additional extensions of the static analysis capabilities of the EEL library include support for V8plus binaries and an improved method for identifying “*hidden routines*”

in the *.text* segment, in the sense that the new method produces much fewer false positives. EEL uses the term *hidden routine* to refer to pieces of code that are neither identified as routines in the symbol table, nor reachable from the nearest preceding routine entry specified in the symbol table. Since compilers include a large number of read-only data in the *.text* segment, there is a high chance that some words of data will have the same binary representation as valid instructions. EEL used to mistakenly consider many such words of data as *hidden routines*. We conduct some more checks before we classify sequences of seemingly valid instructions as *hidden routines*.

While all the extensions enumerated above improve the quality and the accuracy of the results produced by the static analysis, these extensions have a more important contribution at increasing the robustness of the edited binaries produced by the EEL library. A better understanding of the control flow in programs can only result in more stable instrumented binaries.

EEL permits addition and deletion of code at arbitrary locations in the program. Since the size of the text segment cannot be changed as other sections in the binary follow at addresses right after the end of the *text* segment, EEL creates a completely new section named *.edited_text* which is placed after all the other sections in the executable. The new section includes the code from the original *.text* section modified as wanted, and it can grow as large as needed without affecting any other existing sections. The principal challenge for having such a powerful capability, as moving the code to a different location, is to update all the control flow instructions to transfer the control flow to the correct instruction in the new section. For this reason, understanding the original control flow correctly is such a necessary task.

Unfortunately, the target of some indirect jumps cannot be determined using static analysis. For instance, in the “C” language, function pointers can be stored into variables or passed as arguments to other functions. A complex and costly inter-procedural analysis may determine the target of some of these pointers. EEL

implements a simpler solution that uses the space occupied by the old *.text* section to store a translation table. Every instruction in the original section is replaced with an unconditional annulled branch instruction that transfers control to the new location of the instruction. With this mechanism, whenever control flow reaches the old location of a routine entry because of an “unknown” indirect jump, the unconditional branch will transfer the control back to the correct instruction. But all relative branch instructions have a limited range. If the distance between the old and the new address of an instruction is greater than the spanning range of the available unconditional branch instruction, the translation table will contain only the address where the instruction is now located. To accommodate these situations, EEL replaces the indirect jumps whose target cannot be determined, with a code patch that uses the translation table to compute the destination target at run-time.

The above solution works well in practice most of the time. However, occasionally, unpatched code from system shared libraries invokes a routine from the original *.text* segment. Such cases occur in the finalization code of Fortran programs on SPARC machines. We modified the EEL mechanism to handle also such cases. Instead of storing in the translation table the address of the new location of every instruction in the old *.text* section, for routine entries we place a sequence of three valid instructions that act as a trampoline and can transfer control to any place in the address space. Although the trampoline code overwrites the entries corresponding to the second and the third instructions of a routine, we consider that in practice, unknown indirect jumps and control transfer instructions from system shared libraries do not jump to the second or the third instruction of a routine. Until now, we did not encounter any case in which this assumption did not hold.

Finally, we added support in the EEL library to modify the *.dynamic* section of a program to include an arbitrary number of new shared library names that must be loaded when the program starts. No binding of the global symbols from the new libraries is performed by the loader. However, the code in the special initialization

function `_init` is executed when the library is loaded. In the case of a user compiled shared library, the initialization routine can be replaced with custom code. The code in the initialization function can export the addresses of the desired symbols in a special area of the program's address space that is reserved in the instrumented binary. We use this mechanism to force instrumented programs load the `libdl.so` library and export the entry addresses of `dlopen` and `dlsym` routines to the profiling code.

3.2 Static Analysis

Our static analysis subsystem, which includes the binary analyzer as shown in Figure 3.1, is not a standalone application but rather a component of every program in our toolkit. We use static analysis to recover high-level program structure from application binaries, including reconstruction of the control flow graph for each routine, identification of the natural loops in each procedure's control flow graph using interval analysis [30] and determination of loop nesting. In other instances we use static analysis to recover low-level details about an application, e.g. the instruction mix in basic blocks or loop bodies, or the schedule dependencies among instructions. As we describe in the next section, the binary instrumenter uses information from static analysis to determine the most suitable places for adding profiling code.

The next section presents the goals, the strategy and the implementation of our dynamic analysis framework.

3.3 Dynamic Analysis

An application's performance is a measure of how the program utilizes the resources provided by the underneath machine. Characterizing the contribution of the application-centric factors in separation of the target architecture results in models portable across different architectures. Except for program structure information and instruction schedule dependencies that are determined by static analysis, we must observe how

the application exercises the memory hierarchy, determine the instructions' execution frequency, and assess the frequency of synchronization among processes. Although static analysis can provide partial answers to these questions using complex inter-procedural analysis, we need to perform dynamic analysis of the binary to get more complete and accurate results.

We collect histograms of execution frequency during a sequence of computation intervals, information about communication volume and frequency, and histograms of reuse distance seen by each load or store instruction. These factors, combined with information from static analysis, provide a good indication of an application's estimated performance and are independent of any architectural details. For each of these measures, we construct parameterized models by collecting data for a carefully chosen set of input parameters and then fitting models to the data collected.

We use binary rewriting to profile an application's binary and to collect the dynamic data. The static analysis subsystem analyzes each routine by building its control flow graph and performing interval analysis on the graph. The instrumenter uses this information to determine the places where the instrumentation code must be inserted, based on the type of data we want to collect. The following subsections describe in more detail the design and implementation for our data collection infrastructure.

3.3.1 Collecting Execution Frequency Histograms

At each synchronization point in the program, we record the communication partner and the amount of data sent and/or received. Between synchronization points we record a characterization of the computation expressed as a histogram of basic blocks executed. Our aim is to characterize the computation between any two consecutive synchronization points and separately store this data for each synchronization interval. Moreover, the data we collect must be architecture-independent. It is not enough to measure the CPU time or the wall clock time spent in each synchronization in-

terval because the execution time is strongly affected by the hardware on which the application is executed and at the same time it provides no indication of where the inefficiencies are. Inside a synchronization interval we are not interested in a trace of the execution but only in the precise count for each basic block. Thus, collecting the histogram of basic block counts is the reasonable thing to do. We do not need to insert a counter in each basic block in order to measure the precise execution frequency of each basic block. Information sufficient to recover basic block counts and the execution frequency of each control flow graph (CFG) edge can be collected efficiently by placing counters on a subset of selected CFG edges [6].

A routine's control flow graph has the same properties as a flow network. Each directed edge in a control flow graph has an execution frequency, just like each edge in a network flow has a stated capacity. The nodes in the graph are edge junctions. Except for the entry and exit nodes, all the nodes* in the graph have the property that the control flow that enters into the node must equal the control flow that leaves the node. This is the flow conservation property analogous to Kirchhoff's law for the physics of electrical current.

Using the conservation property and the observation that there is no need for more than one counter on a linear sequence of nodes and edges, we can compute the minimum number of counters (NC_{min}) required to recover the execution frequency for the entire graph. In a connected graph the inequality $NE \geq NV - 1$ holds true always. NE represents the number of edges in the graph and NV is the number of vertices. I consider only control flow graphs with one entry and one exit nodes. If the graph has multiple entry or exit points, a simple transformation of adding one absolute starting node with edges flowing into each of the initial entry points and

*This is true in the vast majority of cases. Some programming mechanisms, such as `setjmp/longjmp` family of functions in "C", do not have the conservation property. However, these mechanisms are used in handling exceptions or unexpected errors and are seldom encountered in scientific codes.

a similar transformation for the exit node will ensure that any control flow graph has only one entry and one exit nodes. The static analysis subsystem automatically normalizes the control flow graphs to have this property. For such a graph, only a sequential list of nodes has $NV - 1$ edges. Adding an invisible edge from the exit node to the entry node [6], the conservation property holds for all the vertices in the graph, including the entry and exit nodes. Having the conservation property for the entry / exit nodes makes sense because the number of times the control enters into the routine must be equal to the number of times the control leaves the routine.

Knuth and Stevenson [17] demonstrate that a placement of the counters such that the set of edges without a counter does not create any cycle in the undirected graph, is a necessary and sufficient condition for any solution to the problem of measuring edge and node execution frequency using edge counters. As Ball noticed in [6], a spanning tree of a control flow graph has the maximum number of edges that do not contain a cycle. Therefore, the set of edges that are not part of the spanning tree corresponds to the minimum number of counters needed to profile the entire graph. The number of edges in any spanning tree of a connected control flow graph with NV vertices is $NE_{Tree} = NV - 1$. The minimum number of counters needed is equal to the number of edges that are not part of the spanning tree:

$$NC_{min} = (NE + 1) - NE_{Tree} = NE - NV + 2.$$

The spanning tree is computed on the graph extended with an edge from the exit node to the entry node. For the simplest possible graph, consisting of a sequential list of nodes, $NE = NV - 1$ and therefore, one counter suffices to recover the execution frequency of all the nodes and edges in the graph.

An undirected graph with cycles does not have a unique spanning tree. As a result, the problem of placing $NE - NV + 2$ counters on NE edges does not have a unique solution. From the set of possible solutions we must select the configuration that minimizes the run-time overhead. The run-time overhead is given by the number of extra instructions the profiled binary must execute. Because each counter executes

practically the same code, we have to minimize the number of times any counter is executed. This is equivalent to placing counters on the edges with a low execution frequency, or in other words, avoiding placing counters on the most executed edges. We already know that the sub-graph represented by the edges without a counter must not contain any cycle, therefore, the optimal solution is to build the maximum spanning tree of the control flow graph and to place counters on the edges that are not part of the tree.

Not all theoretical combinations for placing the counters are possible in practice due to technical limitations. Some of the edges in the control flow graph cannot be easily instrumented, meaning that allowing code insertion on them would create too many problems for rewriting a valid, working executable. The theoretical solutions that contain counters on unmodifiable edges are avoided from the start. Because counters are inserted only on edges that are not part of the spanning tree, the solution consists in initializing the tree with the uneditable edges plus the edge from the exit node to the entry node that does not exist in the initial graph. If the set of uneditable edges contains a cycle, the problem of profiling for the execution frequency by placing counters on selected edges does not have a valid solution. In practice such a scenario is not possible.

Collecting the exact computation cost between two synchronization points for a parallel application raises another restriction. Whenever a subroutine is invoked, that subroutine might execute a communication primitive either directly or indirectly by one of its own callees. In such a case, the current computation interval is closed and another interval begins. But we must determine the exact count of the blocks and edges that are executed in each interval. One solution is to insert enough counters in the graph to allow us to recover the execution frequency independently on each side of the call instruction. We can think of a subroutine call as a point where the control flow leaves the current routine and it may return during the same interval or not. Because a call instruction is a point where the control flow leaves the routine,

we can include the call instruction in the set of exit nodes and consider there is a virtual edge from the node of the call instruction to the exit node. By adding this virtual edge to the graph and including it in the set of initialization edges for the spanning tree, we ensure that all the edges incoming into or outgoing from the node of the call instruction are measured independently. Each virtual *call-to-EXIT* edge that is included in the initialization set of edges for the spanning tree, forces one control flow edge that otherwise would connect the “call” node to the rest of the tree not to be included in the spanning tree. As a result, each call instruction increases the number of inserted counters by one. The formula for the minimum number of counters needed to solve the problem of profiling for the execution frequency of all the blocks and edges in the control flow graph, becomes:

$$NC_{min} = NE - NV + 2 + num_calls. \quad (3.1)$$

After all the uneditable edges and *call-to-EXIT* edges are included in the spanning tree, we can apply Kruskal’s algorithm [10] to build the maximum spanning tree[†] (MST) of the control flow graph (CFG). In order to compute the MST, we have to augment all edges with a weight value. For the weight value of an edge we can use either its execution frequency measured during a previous execution of the program, or an estimative execution frequency computed with some heuristic.

We implemented a simple heuristic algorithm to compute the weight values of all the edges based on four basic rules:

- the entry node in the CFG has a weight value of one;
- the weight value of a vertex is divided equally among all its outgoing edges if none of these edges is an exit loop edge;

[†]*Maximum spanning tree* is the short form for the term *maximum-weight spanning tree* which represents the acyclic subset of edges that connects all vertices of the CFG and whose total weight is maximized.

- each loop has a multiplicative factor equal to ten;
- the weight of a node is the sum of the weights of its incoming edges.

In addition to these four rules, a separate algorithm handles the exit edges and the nodes in which the exit edges originate. A CFG edge whose head node is part of a loop and its tail node is outside the loop is an exit edge. In most cases exit edges have their tail node in the program scope immediately outside the one that contains its head node. Still, we encountered cases in which the exit edge crosses several levels of a loop nest. To accommodate these cases, we apply the following algorithm to compute the estimated weight of an exit edge (e_i):

1. determine the outermost loop (L) for which this edge is an exit edge;
2. find the number (N_{exit}) of edges that exit loop L ;
3. the weight of e_i is the weight of the loop L 's head divided by N_{exit} ;
4. all the other outgoing edges of the e_i 's head node receive an equal fraction of the remainder weight of that node, after the newly computed weight of e_i is subtracted.

The final step consists of placing counters on the edges that are not part of the maximum spanning tree. Figure 3.2 presents a sample control flow graph with one loop and the steps that must be performed to determine the optimal insertion place for the counters. The number of counters inserted is validated by formula (3.1), where $NE = 5$ and $NV = 5$. We count only the edges existent in the original CFG. From Figure 3.2(d) we can recover the execution frequency for all the blocks and edges in the CFG.

$$B1 = B3 = B5 = c1; B4 = c2; B2 = c1 + c2$$

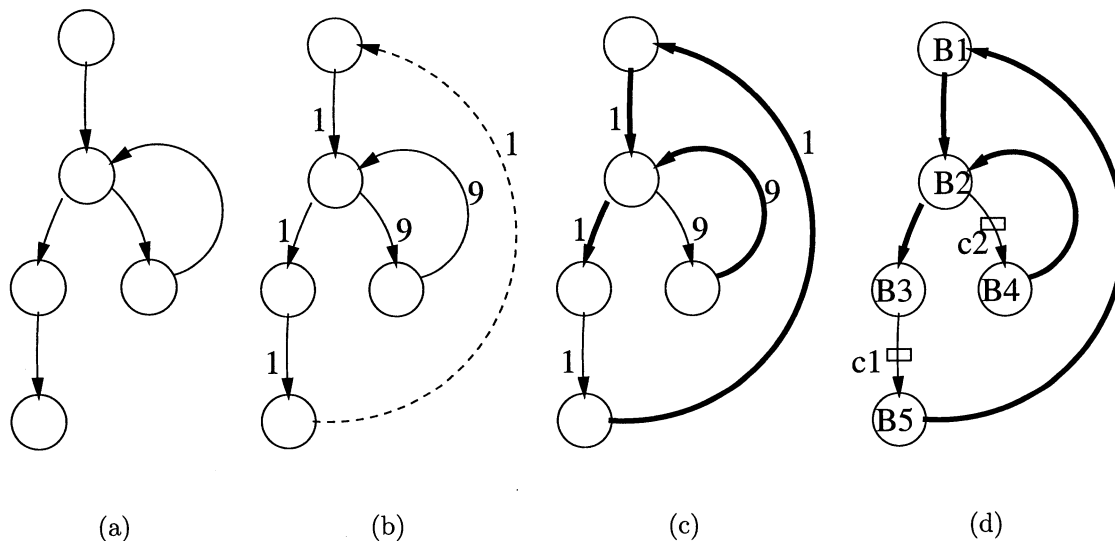


Figure 3.2 : (a) Sample routine CFG; (b) Add an edge from the EXIT node to the ENTRY node and estimate edges execution frequency; (c) Build the Maximum Spanning Tree (MST) of the modified CFG; (d) Insert counters on edges that are not part of the MST.

3.3.2 Collecting Communication Data

In order to model a parallel application, we must capture details about synchronization events. For each event, we collect a snapshot of the stack trace which provides us with the exact location in the program that generated the event. Additionally, we identify the socket or sockets on which messages are exchanged and the volume of data transferred on each socket in each direction during this communication event. A communication event is defined by a call to a communication function. Routines to be considered as communication functions are read from a configuration file passed as an argument to the instrumenter. Enabling the user to specify the communication functions provides more flexibility and results in easy portability to other message passing libraries based on sockets. In fact, the instrumenter has no knowledge about the semantic of the specified functions. It inserts code around all calls to these functions to collect the volume of data transferred on sockets while the execution control

flow entered and did not return from one of them.

Calls to communication functions specified in the configuration file also delimit computation intervals. We collect separate execution histograms for each computation interval. One can collect a single histogram for an entire execution by passing a configuration file with no communication functions defined. Our approach can be used to divide the computation in intervals, not necessarily delimited by communication functions, but by any function call. If the code of the analyzed application is available and can be modified, one can insert a call to a dummy function and specify the name of that function in the configuration file. In this way, with some user intervention, more useful data can be collected sometimes. For example if there is a main execution loop, one can collect data separately for each iteration by inserting a call to an empty function at the beginning or the end of the loop's body. This is useful if the studied application changes behavior over time (e.g. increasing or decreasing amount of work as with algorithms on triangular matrices).

In designing our dynamic instrumentation, we needed to decide which data structure better suits our needs for collecting execution and communication data. An array with one entry for each instrumented counter provides the minimum overhead for finding an entry and incrementing its associated value. However, in an usual application only a fraction of the counters are executed and with the static allocated array of maximum possible size, the space usage is not optimal. The memory overhead is not a very big concern though, because it is proportional to the number of inserted counters and therefore proportional to the size of the analyzed code. The code size is just a (large) constant for a given application (it does not depend on other input parameters). An implementation issue is the need for an extra register required to pass the counter's global index to the event handler routine. The low overhead of this implementation requires a global counter index for fast access into a fixed sized array. The space problem is more significant if we look at the output data file. Saving a full execution frequency histogram with a bin for each counter at each

synchronization point for a program with many communication events, will quickly create a disk space problem. A solution to this problem is to write to disk only the non-zero counters, but this requires traversing the entire sparse array each time we write the data into the file. Another possible solution would be to use an hybrid vector-array data structure. We present a hybrid vector-hashtable solution below.

To solve the extra register problem and also the less important run-time memory overhead problem, we opted for an hybrid data structure consisting of an hash table and a vector, with cross-references between them. The hash table is used for quickly finding the entry for a previously referenced edge or to determine if the counter was never touched before, and the vector structure is used to quickly iterate through all edge counters in the hash table. Each edge counter occupies one entry both in the hash table, based on the result of the hash function, and in the vector, placed in the order they were referenced for the first time.

This hybrid data structure has an $O(1)$ average complexity for *insert* and *find* operations, while traversing the elements and the memory space required are linear in the number of *distinct edge counters referenced*. The overhead of computing the hash key each time a counter is incremented, is larger than in the case of a static array index solution. With the proposed data structure we use the counter's address as input for the hash-function. The address of the edge counter is the address of the *call* to the event handler routine. This information is passed in a default register to the callee function by the SPARC ABI. A *delete* operation is not as efficient with this data structure, but collecting profile data does not require deleting individual elements. The *clearAll* operation has an overhead directly proportional to the size of the data structure, and thus to the number of *distinct executed counters*. The *clearAll* operation is used much less frequently than *insert* and *find* operations, once at every synchronization event when a computation interval ends and another one begins. At this point, the compact data in the vector is written in the output file and the data structures are reinitialized.

3.3.3 Monitoring Memory Access Behavior

During execution we compute a characterization of an application’s memory access behavior. Because we focus on collecting architecture-neutral information only, we decided to capture the pattern the application uses to access the memory. We collect the memory reuse distance [7, 21, 12] seen by each data access during execution. Reuse distance is a measure of the number of distinct memory locations accessed by the program between two references at the same datum.

Characterizing memory access behavior in terms of data reuse in programs has two important advantages. First, data reuse is independent of any cache configuration or architecture details. Second, “data reuse is the main determinant in cache performance because all cache reuse comes from reuse of the same or adjacent data” [12]. Therefore, reuse distance separates application-specific factors from architecture-dependent factors and has all the properties that we want.

We collect reuse distance information separately for each reference in the program. Before each memory reference we invoke a library routine that augments a histogram of reuse distance values for the reference. In addition to the address of the reference, the event handler needs to know the address of the memory location referenced and the number of bytes touched by this instruction. Our implementation of the event handler computes a compressed form of the complete histogram of reuse distances seen by each memory reference. To compress the size of the output data, we coalesce bins with near distances before the data is written in the output file. The small loss of detail has no noticeable effect on the precision of the models of reuse distance that we construct, but the reduction in space is often significant. The event handler routine uses a hash table to assign a logical timestamp to each memory block referenced by the program. The timestamp enables us to determine the reuse distance between a pair of accesses to the same datum. The hash table stores only the timestamp of the last access to each block. This data structure would be enough to count how many memory locations, distinct or not, were referenced since the last access to the

same datum. To determine the number of **distinct** memory locations accessed, we use a balanced binary tree with one entry (node) for each memory block referenced by the program. The sorting key of the balanced search tree is the logical timestamp associated with the last access to that location which we obtain from the hashtable. By using a unit size memory block we collect pure temporal reuse distance which cannot account for the spatial reuse in the cache lines. However, by setting the memory block to be equal with the cache line size, we can also measure the spatial reuse in a cache line because we collect the reuse distance of data blocks rather than data elements. The size of the memory block our runtime library uses is defined by an environment variable; therefore collecting data for different cache line sizes does not require re-instrumenting the code or re-compiling the event handler routine.

Our implementation of the event handler executes the following pseudo-code for a memory access to the location $addr_i$ by the program instruction $inst_k$:

step 1 Generate a logical timestamp t_i^{new} equal to the next available value of the *global timestamp counter*, and increment the global counter.

step 2 Search the hash table for the memory block b_i with address $addr_i$ (complexity $O(1)$). If the address is not found, then this memory access corresponding to the load or store instruction $inst_k$ is the first reference to block b_i ; increment the number of cold misses seen by instruction $inst_k$, and insert a new entry (key= $addr_i$, value= t_i^{new}) into the hash table (all operations are in constant time); then go to **step 5**. If $addr_i$ is found in the hash table, then block b_i was touched before by this or another instruction; go to the next step.

step 3 Read the timestamp t_i^{last} corresponding to the previous access to b_i , and replace into the hash table the value corresponding to block b_i with the newly generated timestamp t_i^{new} .

step 4 Delete the node with key t_i^{last} from the binary search tree. While searching for the node to be deleted, we count the number of nodes (NN) with the key

greater than t_i^{last} . NN corresponds to the number of distinct memory blocks that have been referenced since the previous access to b_i . Each node of the tree maintains a field *size* representing the number of nodes in the sub-tree rooted at it. For a binary search tree with the greater keys on the right, NN is a sum of terms $(node_j.size - node_j.leftChild.size)$, for every $node_j$ with the key greater than t_i^{last} that was encountered on the path from the root to the node to be deleted. The delete and count operations have an aggregate $O(\log N)$ time complexity, where N is the number of distinct memory blocks touched by the application until that moment. Next, record that memory instruction $inst_k$ saw an access with reuse distance NN .

step 5 Insert a node with key t_i^{new} into the binary tree. This step has complexity $O(\log N)$.

Time complexity for computing the reuse distance seen by one memory reference is $O(\log N)$. Overall, the overhead of collecting memory reuse distance information for the entire execution is $O(M \log N)$ and the memory space required by the data structures for monitoring reuse distance is $O(N)$, where M is the number of dynamic memory instructions executed. Time and space complexities are both significant even with these optimized data structures. Collecting memory reuse distance is much more expensive than collecting the histogram of edge counters.

3.3.4 General Features of the Binary Instrumenter

The Executable Editing Library (EEL) [18], on top of which our binary analyzer and instrumenter are built, provides support for insertion of static machine code into the binary. To avoid the need to re-instrument an application binary each time we want to change the way instrumented events are processed, we added support for enabling instrumented code to invoke routines provided in a dynamically loaded shared library. In the initialization part of the modified binary, we added support for opening a shared

library and locating symbols in it. We locate the appropriate symbols in the shared library and bind them to small stub routines to which the instrumenter statically inserts calls into the edited binary. The name of the library, as well as the symbol names to be searched, are passed as arguments to the instrumenter, and the edited binary uses the provided names. This approach enables the user to supply a new shared library which can process the events differently. To ensure it is used, a user have to include the new library's location in the `LD_LIBRARY_PATH` environment variable, eventually before other libraries that have an identical name. Therefore, our edge counters are in fact calls to the stub routines which in turn will invoke the corresponding code from the shared library which is determined only at run-time.

Our infrastructure for dynamic execution analysis also supports selective or partial execution profiling. Selective profiling is provided by the optional use of *include file* and/or *exclude file* arguments. If an *include file* is provided, only the routines specified in that file are profiled. If an *exclude file* is specified, the routines listed in this file will not be profiled. Partial execution profiling is accomplished by inserting calls to two special routines provided in a small library. These calls can be inserted only in the application's source code. The instrumenter detects the presence of these hooks in the binary and adds the necessary code to start or to stop the collection of data. Starting and stopping data collection is implemented by dynamically modifying the code of the event handling trampolines to either forward the events to the actual event handler routines or to return immediately without processing the events. Although this approach requires access to the source code and partial recompilation and linking of the binary, we found it to be very useful. One can exclude initialization and finalization steps from data collection, or collect information only for particular fragments of code at specific times during the execution (including the routines called from those particular pieces of code). The *start collect* and *stop collect* handlers are executed as any other ordinary code and calls to them can be guarded by conditional statements, effectively enabling data collection to be started and stopped

under arbitrary conditions.

The instrumentation infrastructure that we developed for collecting data about program behavior is quite flexible and can be easily adapted to collect other information or to perform different types of online analysis.

3.4 From Data to Parameterized Models

Though the mechanics of data collection are somewhat complex, assimilating the collected data into accurate models of application performance is a fundamentally more difficult undertaking.

The goal of this work is to build parameterized models that enable us to predict performance for data sizes that we haven't measured. To achieve this goal, we first collect data from multiple runs with different and determinable input parameters. Input parameters should include a numerical measure of the problem size and the number of processors used during the execution.

We compute a parameterized curve for a program variable by using the collected data of multiple runs in which that parameter is modified and all the others are maintained constant.

3.4.1 Building models of the execution frequency

A program's performance is a function of application characteristics and resources available on the target architecture. An architecture-neutral model considers only the application-specific factors. The most important architecture-neutral factors that influence the performance of a single node execution are the number of instructions executed, the mix of instructions, the instruction schedule dependencies in the most frequent executed loops, and the memory access pattern exhibited by the application.

To build models parameterized by the problem size[‡] for the execution frequency of each edge counter, we use quadratic programming [20] on the edge counter's exe-

[‡]The tool can be used to build a model in any input parameter, not only problem size. It requires

cution frequency data, collected from multiple runs with different problem sizes. We use a modeling strategy implemented in Matlab to determine the function that best approximates the input data we collected. The approximation function is written as a linear combination of a set of orthogonal basis functions. The program uses either a default monomial base or a set of user-provided basis in symbolic form so that logarithmic or other non-linear contributions to the model can be considered. The modeling program computes the coefficients of the basis functions in the linear combination that closest approximates the collected data. We include restrictions to reduce or remove oscillations of the resulting fit and to ensure that the computed function is either convex or concave, depending on the program characteristic that is modeled. Our approach works best with scientific codes that have predictable execution patterns, namely, ones that do not use adaptive algorithms. For such applications, we show that we can compute accurate models even in multiple parameters. For adaptive algorithms, we can produce an approximate model with reasonable accuracy for one parameter models.

The data files collected with our infrastructure must be initially processed by a filter program which outputs data in a format understood by our Matlab scripts. The filter can be configured to output either counters frequencies or the number of executed instructions. A set of command line arguments controls the output in each case. For example, counters can be sorted either by location (grouped by routines) or by frequency with the most executed ones first. Optionally, the output can be limited to a most significant threshold. Instructions are classified by type and can be aggregated at any level in the scope-tree of the program. We defined a set of generic RISC instruction classes and a module for translating native SPARC instruction into generic RISC instructions. The filter computes the number of executed instructions for each generic class and each basic block in the program. Using static analysis of the

multiple runs where the desired parameter is varied and the others are maintained constant or are a fixed linear transformation of the varied parameter.

binary, the filter builds a scope-tree that reflects the program structure [24]. There are three possible scopes that can be used to describe the program structure:

- *Program* scope - is the root of the tree and its children are *routines*;
- *Routine* scope - is on the second level in the tree and its children are *loops*;
- *Loop* scope - can include any other number of *loops*.

If the binary contains debugging information, the routine and loop scopes are annotated with source file information, including the name of the source file and the range of line numbers corresponding to that scope in the source file. If the binary contains source line number information, the filter performs a normalization step which attempts to fold together the information for loops that have overlapping ranges of source line numbers, and the same parent in the scope-tree. Such loops are the result of compiler optimizations, including loop fission, software pipelining, loop-invariant code motion, or tiling. The instruction count can be outputted at program level, at routine level or at individual loop level. For example, if we select to output instruction counts at routine level, we can build a separate model for the execution frequency of each instruction type and each routine. In all the above cases, the filter can receive the optional arguments *include file* and/or *exclude file* having the same semantic as for the binary instrumenter (see Section 3.3). All these options make the filter a flexible tool for transforming the edge counters data into other forms from which it can be modeled.

3.4.2 Building models of the memory access pattern

We use a similar approach to model the memory access pattern exhibited by the application so we can predict the memory hierarchy response of the application on any target architecture. Using dynamic analysis, we collect the memory reuse distance seen by each memory reference. Temporal reuse distance is a measure of the number of distinct memory locations referenced between a pair of accesses to the same datum.

Spatial reuse distance is a measure of the number of distinct locations referenced between a pair of accesses to the same cache line. Temporal reuse distance is a totally architecture-independent measure, while the spatial reuse distance depends only on the length of cache lines.

For a fully-associative cache, we can predict if a memory access is a hit or a miss by simply comparing its reuse distance with the cache size (see Figure 3.3). Beyls and D’Hollander [8] show that reuse distance predicts the number of cache misses accurately even for caches with a low associativity level or direct mapped caches. However, a model based on reuse distance alone cannot predict conflict misses.

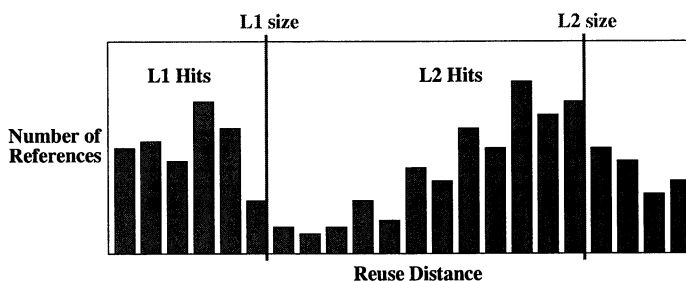


Figure 3.3 : Example of reuse distance histogram. All references with reuse distance less than the cache size are hits.

We use reuse distance data to model the behavior of each memory instruction and to predict the fraction of hits and misses for a given problem size and cache configuration. Our first attempt at predicting if a memory instruction is a hit or a miss for a given problem size tried using the average reuse distance for that instruction. We were able to produce a very accurate parametric model for the average reuse distance of memory accesses in Sweep3D (see Figure 3.4), a three-dimensional particle transport problem used as an ASCI benchmark for evaluating high performance parallel architectures [1]. However, this model proved inaccurate for predicting the number of misses when we compared against results measured during a run on an SGI Origin 2000. On the Origin, we measured L1 and L2 cache misses during execution using

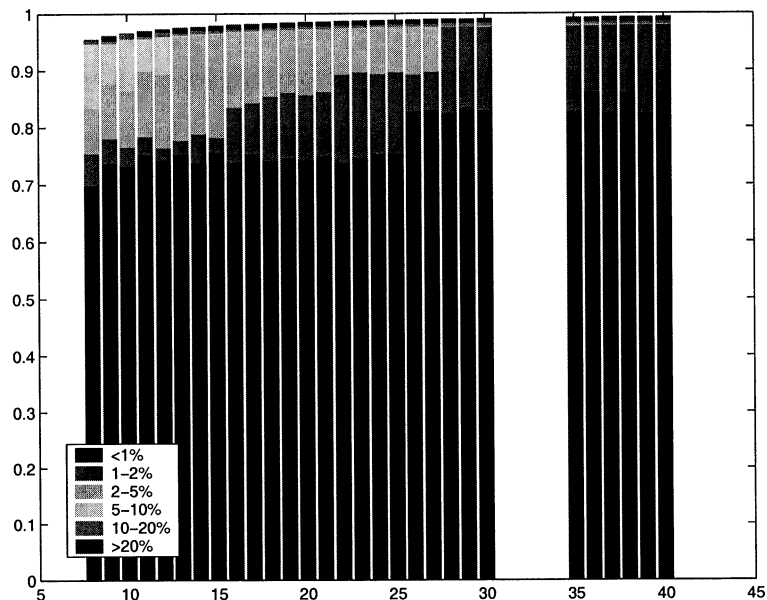


Figure 3.4 : The error distribution of the average memory reuse distance model for Sweep3D. The x axis represents the problem size and the y axis represents the fraction of accesses. The model has less than 1% error for three quarters of the accesses and the tendency is to have a higher precision for larger problem sizes.

hardware performance counters. After analyzing the discrepancies between measured and predicted values, we realized that most memory instructions had a dual behavior: only a fraction of the accesses by a load/store accessing a stride one sequence were misses while our model was predicting either 100% hits or 100% misses. By considering only the average reuse distance when forming our model, we lost a significant part of the information that we collected. One very large reuse distance averaged with several very short reuse distances results in a larger than cache size average, which yields a prediction of all misses.

Based on our experience, we adopted a new approach for modeling the dynamic behavior associated with a load or store, by using a compressed form of the complete histogram of reuse distances collected by our binary instrumentation infrastructure. The modeling strategy first divides the histogram of accesses for an instruction into multiple bins and then computes a separate parameterized model for each bin. The

algorithm for determining the number of bins and their size and structure uses a divide and conquer approach, recursively splitting a set of accesses in two until the two subsets have similar fitting curves. We apply the algorithm to the entire set of data and at each step we execute a similar division for the data sets corresponding to each problem size. We use a heuristic algorithm to determine how to partition the accesses. Its decisions influence the convergence speed, the accuracy and the stability of the final model. In our experiments, the partitioning heuristic that yielded the most stable and accurate results was one that selects partition boundaries so that the ratio between the number of references in the two partitions resulting from a split are similar across all problem sizes. After partitioning, we perform a coalescing step that examines adjacent bins and aggregates them together if they have similar polynomials describing them. Each bin is modeled by two polynomials, one that models how the number of accesses which are part of that bin changes with problem size and one that models how the average reuse distance of these accesses changes with problem size.

Figure 3.5(a) shows the reuse distance histogram data collected by our tool for one of the most frequently executed memory accesses in Sweep3D. Figure 3.5(b) presents our parameterized model for that instruction. On each graph, the x axis represents the problem size (from 8 to 46 in this case); the y axis represents the normalized number of accesses for each problem size; and the z axis represents the reuse distance. Our model, parameterized by problem size, can be evaluated for an arbitrarily large problem size in a fraction of a second. The problem of determining the ratio of hits and misses for a given cache size $csize$ is equivalent to determining the intersection of the model with the plane defined by $z = csize$. Similarly, the problem of computing the expected behavior for one instruction at a given problem size $psize$ is equivalent to determining the intersection of the surface and the plane defined by $x = psize$. We can also formulate the problem of determining the minimum cache size such that the hit-ratio is h . The solution to this problem is the intersection of the built model and the plane defined by $y = h$. Any two of these three problems can be combined and

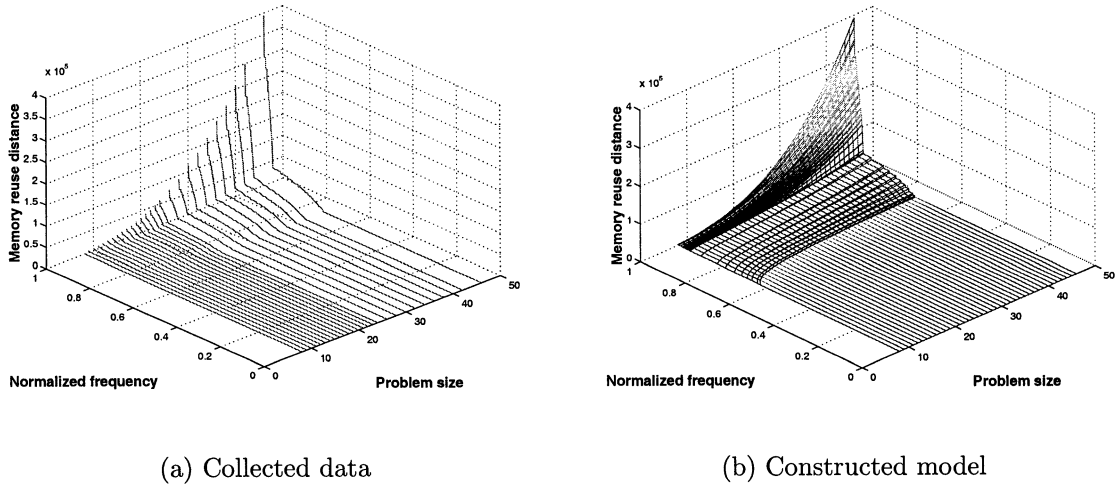


Figure 3.5 : The surface on the right is the parameterized model for one of the most frequently executed memory accesses in Sweep3D. On the left is the original data collected for that instruction. The x axis represents the problem size, on the y axis is normalized number of accesses and the z axis represents the reuse distance.

the solution is the intersection of the surface with the corresponding two orthogonal planes.

The actual problem that we want to solve is predicting the ratio of misses for a given problem size and cache size.

For the memory reference whose access characteristics are reflected in Figure 3.5, approximately 75% of the executed accesses see a small, constant memory reuse distance and therefore will be hits for any problem size. The other 25% of the accesses end up in several bins, each one having a distinctive monotonically increasing function for the reuse distance. The constant reuse distance seen by three quarters of the accesses is due to the spatial reuse in the cache. The cache line considered in the model is 32 bytes long and four elements (of type `double`) can be packed in one cache line. The first access to a cache line results in a large reuse distance and the next three stride-1 accesses to the same cache line have a small reuse distance.

Figure 3.6 illustrates how an instantiation of the memory reuse distance model

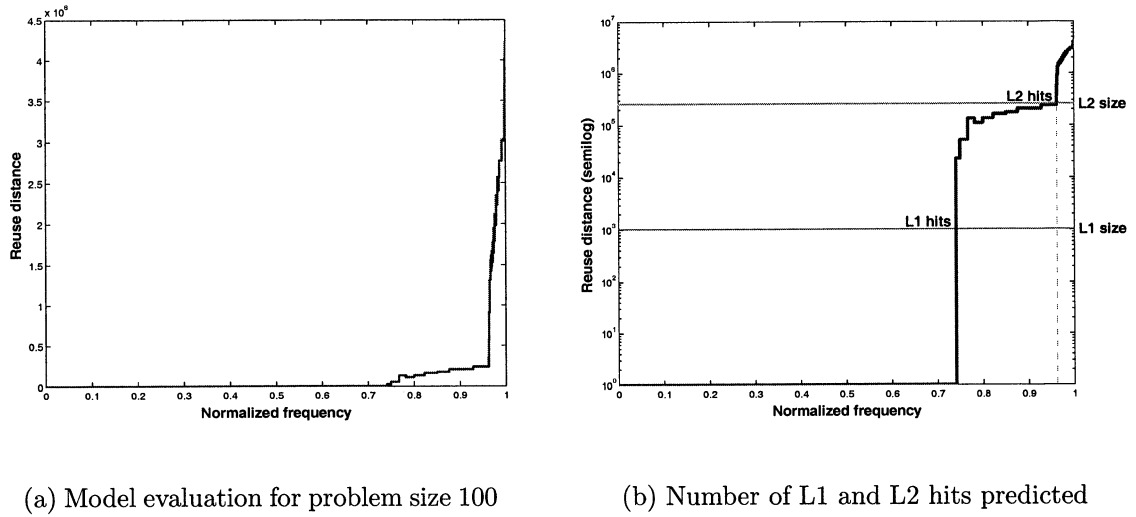


Figure 3.6 : An instantiation of the model in Figure 3.5(b) for problem size 100. The curve on the left presents the histogram of reuse distances in linear coordinates. The curve on the right is shown on a logarithmic y-axis and includes the cuts for the L1 and L2 cache sizes.

presented in Figure 3.5(b) translates into a prediction of cache misses. The model is evaluated for problem size 100 and we consider an architecture with 1024 L1 blocks and 256k L2 blocks. Because the maximum reuse distance predicted for this reference is six orders of magnitude larger than the size of the L1 cache, the curve on the right is shown on a logarithmic y-axis. The model predicts a ratio of about 74% hits in the L1 cache and 96% hits in the L2 cache.

Many instructions have a more uniform distribution of their accesses' reuse distance. Figure 3.7 presents another frequently executed instruction from Sweep3D. For this memory reference, about three quarters of the accesses have a small, constant reuse distance. In this case, the other 25% of the accesses are grouped in a single bin with a linear growth function for the reuse distance.

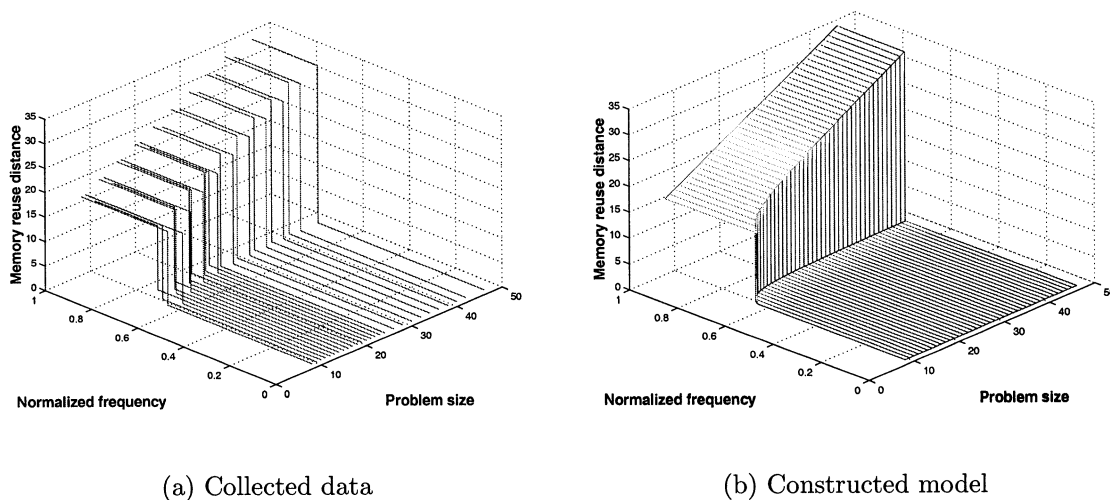


Figure 3.7 : The collected data and the constructed model for another frequently executed memory access from Sweep3D. The x axis represents the problem size, on the y axis is normalized number of accesses and the z axis represents the reuse distance.

3.5 Mapping the Models to a Target Architecture

Our post-processing tool constructs an annotated control flow graph (CFG) that contains information about loop nesting structure and the execution frequency of each basic block. We build this annotated CFG for an application using static analysis information gathered from the application binary along with dynamic measurements of its execution behavior or with an instantiation of the parameterized model. From this representation, we identify paths in the control flow graph and compute their associated frequencies. Inside nested loops, we work from the inside out; no basic block is considered at more than one loop level. These paths serve as input for an instruction schedule analysis tool that computes an estimate for the execution cost of each path.

To compute the execution cost associated with a path for a (possibly different) target architecture, we translate the instructions present in the basic blocks of the (SPARC) application binary into instances of generic RISC instruction classes. We

defined a set of generic RISC instruction classes and a module for translating SPARC binary instructions into generic instructions (see Table 3.1). We built a configurable scheduler that is initialized with an architecture description that enables us to compute predicted execution times for the specified target architecture. The architecture description defines the number and type of execution units, and a characterization of each generic instruction class in terms of hardware resources required. Each generic RISC instruction has an associated set of execution units onto which it can be scheduled, a latency and a repeat rate. The initial scheduler prototype considers each memory access as a primary cache hit and it does not consider the fetch/decode stages. An ongoing effort is focused on translating our data on memory reuse distance into an estimation of latency for a given target memory hierarchy. Currently, we use SPARC-based computers from Sun to collect the data and analyze the binaries; we use the scheduling tool to predict performance on a MIPS R12000 processor; and we validate our predictions against actual executions on a MIPS R12000 by collecting performance measurements with hardware counters.

Native SPARC Instruction	Generic RISC Class
SETHI, ADD[X][cc], TADDcc[TV], SUB[X][cc], TSUBcc[TV], SAVE, RESTORE	IB_int_add
BIcc, BPcc, FBfcc, FBPfcc	IB_br_CC
BPr	IB_branch
CALL, JMPL, RETT	IB_jump
AND[N][cc], OR[N][cc], X[N]OR[cc]	IB_logical
SLL, SRL, SRA	IB_shift
MULScc, UMUL[cc], SMUL[cc]	IB_int_mult32
MULX	IB_int_mult64
UDIV[cc], SDIV[cc]	IB_int_div32
UDIVX, SDIVX	IB_int_div64
TICC	IB_trap
MOVR, MOVcc, RDCCR, RDASR, RDPSR, RDWIM, RDTBR, RDY, WRCCR, WRASR, WRPSR, WRWIM, WRTBR, WRY	IB_int_move
FADD{s,d,q}, FSUB{s,d,q}, FCMP[E]{s,d,q}	IB_fp_add
FDIVs	IB_fp_div32
FDIV{d,q}	IB_fp_div64
FMUL{s,d,q}, FdMULq, FsMULd	IB_fp_mult
FMOV{s,d,q}, FABS{s,d,q}, FNEG{s,d,q} FMOVR{s,d,q}, FMOVcc{s,d,q}	IB_fp_move
FSQRTs	IB_fp_sqrt32
FSQRT{d,q}	IB_fp_sqrt64
F{s,d,q,i,x}TO{s,d,q,i,x}	IB_fp_cvt
LDSB, LDSH, LDUB, LDUH, LD, LDD, LDSBA, LDSHA LDUBA, LDUHA, LDA, LDDA, LDX, LDFSR, LDCSR	IB_load_gp
LDF, LDDF, LDC, LDDC	IB_load_fp
LDSTUB, LDSTUBA, SWAP, SWAPA	IB_load_atomic
STB, STBA, STH, STHA, ST, STA, STD, STDA, STX, STFSR, STCSR	IB_store_gp
STF, STDF, STDFQ, STC, STDC, STDCQ	IB_store_fp
PREFETCH, PREFETCHA	IB_prefetch

Table 3.1 : Classification of SPARC native instructions into generic RISC classes.

Chapter 4

Experiments

This chapter describes several test cases in which our performance toolkit have been applied. The following sections present how we constructed parameterized models of the floating point operation (FLOP) count for two applications, PSTSWM and CRM, and a test case in which we predict the cache miss ratio for the main computational routines in PSTSWM. We briefly describe each of these applications in their respective sections.

4.1 Characterizing the PSTSWM Application

PSTSWM is a message-passing benchmark code and parallel algorithm testbed from Oak Ridge National Laboratory(ORNL). PSTSWM solves the nonlinear shallow water equations using the spectral transform method.

4.1.1 Constructing FLOP Count Models for PSTSWM

When Pat Worley, a scientist at ORNL, heard of the capabilities of our modeling toolkit, he requested that we use it to build a parameterized model for the lower bound of the number of floating point instructions executed by the PSTSWM's main computational loop. The initialization and post-processing phases should not be considered in the model.

In order to measure only the computational part of the main loop of PSTSWM, we placed the *start_collection* and *stop_collection* special markers immediately before and after the main loop in the code. The PSTSWM application reads a configuration file containing several input parameters. Most of them have only a multiplicative

effect on the number of iterations executed. There are only two arguments that can be independently modified which have a non-linear effect on the number of instructions executed. These are the parameters that must appear as symbolic terms in our model. One of these two variables, called $NLON$, can be any strictly positive natural number that has prime factors in the set $\{2, 3, 5\}$ only, and it must always be divisible by four. This restriction is caused by a Fast Fourier Transformation module used in the computation. The other significant variable, MM , can be any strictly positive natural number that is less than one third of the value used for $NLON$.

We started by constructing a model in one variable. Because MM is upper bounded by $NLON$, the only option was to parameterize the model by $NLON$ and to restrict MM either to a fixed value or to a fixed linear transformation of $NLON$. The most important set of problem cases, corresponding to a triangular truncation of the spectral coefficients, arises when MM takes the largest value possible for a given $NLON$. Therefore, for any valid $NLON$, MM 's value is computed by the formula $\lfloor \frac{NLON-1}{3} \rfloor$.

We compiled the PSTSWM application with the Sun WorkShop 6 update 2 FORTRAN 95 6.2 compiler using the $-O4$ optimization flag. Using our instrumentation infrastructure, the optimized binary was augmented to collect edge counter values for the entire program, without any synchronization point defined, and using the special inserted markers to start and stop the collection of data whenever entering or exiting from the main computation loop. To automate the collection of data we wrote a script that for each given $NLON$ value, computes the other program arguments, generates the input file and then executes the instrumented binary.

We used the filter tool to process the collected data files and compute a count for each instruction type for the entire execution. The focus of our analysis was on the floating point operations, as stated by the problem's requirements. We did not consider operations loading/storing floating point values from/to memory, or the register copy operations for floating point values. Additions and multiplies are almost

the only operations performed on floating point numbers; we build separate models for the frequency of adds and multiplies. Without any additional information about the analyzed application, we used a monomial base for the first attempt at building the models. Figure 4.1 shows the number of floating point add operations, measured by our tool in a sequence of executions for different values of $NLON$. For small intervals of $NLON$, the instruction count oscillates, while the general shape of the curve shows rapid growth in the number of executed instructions as $NLON$ increases.

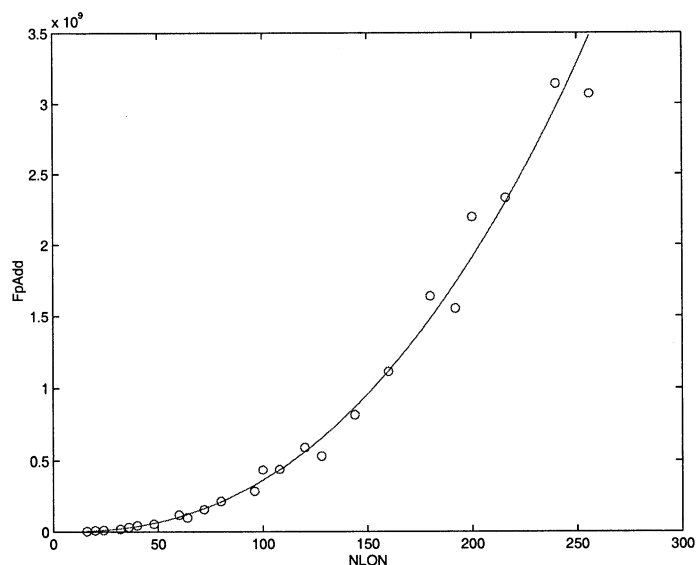


Figure 4.1 : The number of FpAdd operations measured for all valid $NLON$ values less than 256. The measured number of executed instructions does not follow a smooth curve.

This oscillatory behavior is caused by the Fast Fourier Transform module, which is more efficient for problem sizes that are powers of two and least efficient for problem sizes that are divisible by 5. Consequently, when building further models we limit our study to problem sizes that are powers of two. Even in this case, the FFT computation is more efficient when the number of points is a power of four than for values that are powers of two but not powers of four (see Figure 4.2).

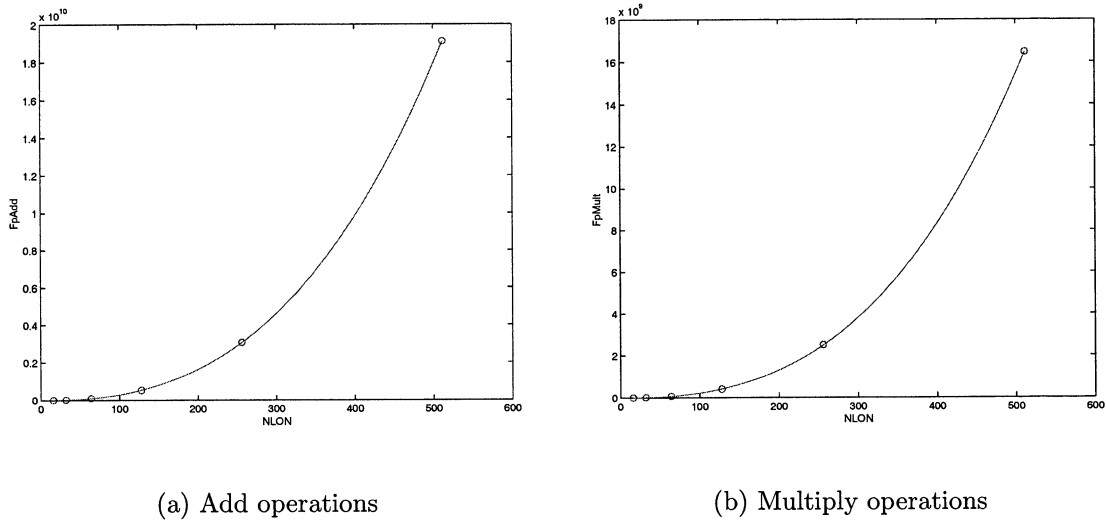


Figure 4.2 : The number of FpAdd and FpMult operations, measured for $NLON = 2^k$, $k = \overline{4, 9}$.

Using only the six data points corresponding to $NLON$ values that are power of two, and the default monomial base, our toolkit computed the following one parameter models for the number of FpAdds and FpMults executed during the entire computation in the main loop and routines it calls:

$$FpAdd = 100.25 * NLON^3 + 22235 * NLON^2 - 332905 * NLON + 3364329 \quad (4.1)$$

$$FpMult = 95.6 * NLON^3 + 14328.9 * NLON^2 - 228246 * NLON + 2647343 \quad (4.2)$$

Our next step was trying to derive a model in two variables. For the one parameter models, we restricted MM to be about one third of $NLON$. Now, we want to vary MM and $NLON$ values independently to see if we can come up with a model in two variables. For each $NLON$ value in the set $\{16, 32, 64, 128, 256, 512\}$, we executed the profiled binary several times, varying MM in the interval of admissible values between one and $\lfloor \frac{NLON-1}{3} \rfloor$. For each $NLON$ value we built a model in MM using the default polynomial base. The resulting models are all quadratic in MM and the fitting error computed by our tool is zero. For each of the six $NLON$ values and for each instruction type, the model in MM has the general formula: $a_2 * MM^2 + a_1 * MM + a_0$.

Thus, for each considered instruction type, we have six models, one for each distinct $NLON$ value. The six models corresponding to an instruction type, have the same general form. Each coefficient from the general formula presented above, can be written as a model in $NLON$. To do so, we apply the same modeling strategy as the one described before, using $NLON$ as the varying factor and the coefficients a_2 , a_1 , a_0 as the measures to be modeled. Next, the coefficients of the general formula in MM can be replaced with their computed models in $NLON$. The result is a model in two variables for each instruction type:

$$FpAdd = (843.5 * NLON + 361.5) * MM^2 + (7591.5 * NLON + 1084.5) * MM \quad (4.3)$$

$$+ (4.64 * NLON^3 + 21539.4 * NLON^2 - 637926.64 * NLON + 16925208)$$

$$FpMult = (843.5 * NLON + 2169) * MM^2 + (5844.25 * NLON + 6507) * MM \quad (4.4)$$

$$+ (14040.8 * NLON^2 - 546619 * NLON + 18149057)$$

All of the modeling work for this experiment up to this point, was based on the default monomial base. Usually the analyst who is working with an application and wants to model its performance, has an idea about the algorithms used and what their complexity terms may be. We know that the application uses an FFT library, and that Fast Fourier Transformation has a logarithmic term describing the scaling of its performance. We recomputed our models using a custom base that includes not only the usual monomic terms but also a logarithmic term. With this basis set, our modeling tool produces the following models in two variables:

$$FpAdd = (843.5 * NLON + 361.5) * MM^2 + (7591.5 * NLON + 1084.5) * MM \quad (4.5)$$

$$+ (2255.7 * NLON^2 \log(NLON) + 2438.54 * NLON^2$$

$$- 3093.68 * NLON + 158417)$$

$$FpMult = (843.5 * NLON + 2169) * MM^2 + (5844.25 * NLON + 6507) * MM \quad (4.6)$$

$$+ (745.4 * NLON^2 \log(NLON) + 6535 * NLON^2$$

$$- 107019 * NLON + 1384697.6)$$

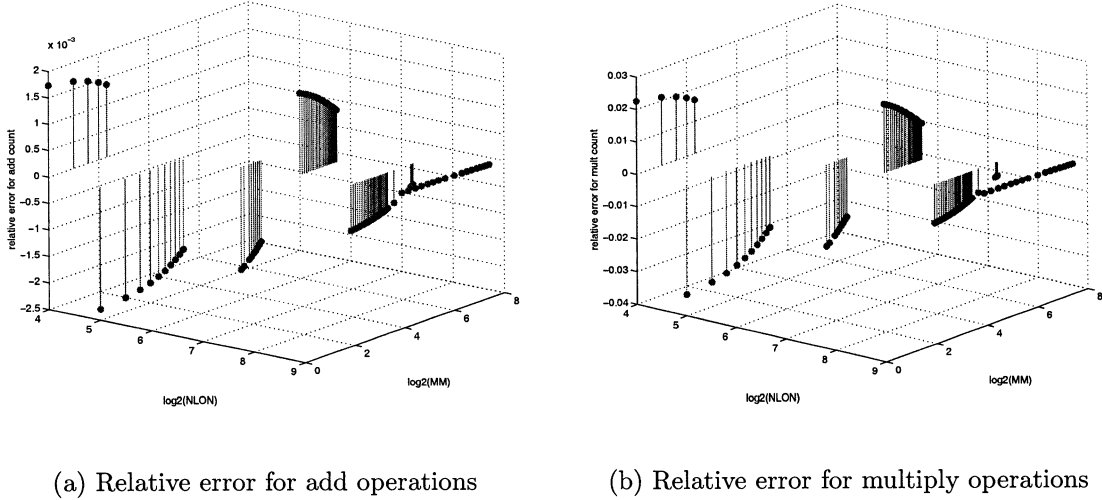


Figure 4.3 : The accuracy of the constructed model in two variables, relative to the measured data collected by our tool.

4.1.2 Validating the FLOP Count Models

This section presents several approaches that we used to verify the accuracy of our synthesized models. First, we verify that our model closely resembles the data collected with our dynamic execution measurement infrastructure. The main purpose of this first step is to validate our modeling tool. From this first experiment we cannot conclude anything about the correctness of the instrumenter or the soundness of the data collection algorithm used.

Figure 4.3 plots the relative error of the models described by the equations 4.5 and 4.6, when compared to the collected data. From the figure we can see that the relative error of the models is less than 0.3% for the FpAdd operations and at most 3.5% for the FpMult count.

The models do not perfectly fit the collected data points because the number of operations in the FFT module cannot be perfectly described by monotonic functions. The varying efficiency of the FFT algorithm has a greater impact on the multiply operation count. Later, when we will build separate models for each routine, we

show that the error of our models is limited to only the two routines that perform the core FFT computation. Although the plots in Figure 4.3 cover only the range of parameter values for which we collected data, we note that the relative error decreases for both add and multiply operation count models when MM gets larger. The FFT computation is independent of parameter MM , so the fluctuating efficiency of the FFT part has a smaller contribution to the overall performance when MM grows and more operations are performed in the other parts of the application.

A second approach for validating the models is to do a hierarchical analysis. We can configure the filter tool to compute the instruction counts at routine level and we can build separate models for each routine. We limit our analysis only at those routines that contain floating point instructions. For PSTSWM there are seventeen routines that execute at least one floating point instruction. Tables 4.1 and 4.2 present the models computed for each routine. It also presents the relative error of the models compared to the row data collected for each routine and instruction type considered.

Routines are listed in the alphabetical order. Except for the first two routines, the models perfectly approximate the measured values. The first two routines are part of the FFT library and we explained before the cause for inaccuracy of our models.

The model for the entire program is the sum of all per routine models. We can check how this hierarchical model compares to the original model computed directly for the entire execution. The aggregated models for the entire program less the first two routines are:

$$\begin{aligned} FpAdd &= (843.5 * NLON + 361.5) * MM^2 + (7591.5 * NLON + 1084.5) * MM \quad (4.7) \\ &+ (4277.25 * NLON^2 + 9881 * NLON + 2892) \end{aligned}$$

$$\begin{aligned} FpMult &= (843.5 * NLON + 2169) * MM^2 + (5844.25 * NLON + 6507) * MM \quad (4.8) \\ &+ (3615 * NLON^2 + 7772.25 * NLON + 7229) \end{aligned}$$

The partial models above perfectly approximate the collected data. The modeling error is constrained at the first two routines only. Separately, we summed the models

for the two FFT routines:

$$FpAdd = (2257.7 * NLON^2 \log_2(NLON) - 1858.2 * NLON^2 - 12240.88 * NLON + 146380) \quad (4.9)$$

$$FpMult = (769 * NLON^2 \log_2(NLON) + 2692.5 * NLON^2 - 106059.6 * NLON + 1268178) \quad (4.10)$$

The models confirm our observation that the FFT computation does not depend on MM . When MM gets larger and $NLON$ is held constant, the relative contribution of the FFT computation to the overall instruction count decreases and so does the modeling error. We can subtract equations 4.7 and 4.8 from the model for the entire program described by equations 4.5 and 4.6 to obtain another formula for the FFT model:

$$FpAdd = (2255.7 * NLON^2 \log_2(NLON) - 1838.7 * NLON^2 - 12974.68 * NLON + 155525) \quad (4.11)$$

$$FpMult = (745.4 * NLON^2 \log_2(NLON) + 2920 * NLON^2 - 114791.25 * NLON + 1377468) \quad (4.12)$$

As we expected, the new models for the FFT module are not exactly equal to the ones described by the equations 4.9 and 4.10, but the differences are not significant. In the hierarchical approach, we were able to isolate the fluctuating instruction counts at those routines that produce it and allowed us to build a clean, precise model for the rest of the program. This experiment proves the viability of our approach and the accuracy of our modeling program.

4.1.3 Memory Reuse Distance Models for PSTSWM

This section describes our attempt at characterizing the memory hierarchy behavior for the main computational routines of the PSTSWM application.

To characterize the memory hierarchy behavior of PSTSWM, we compiled the application for the v8plus SPARC ABI using the Sun WorkShop 6 update 2 FORTRAN

95 6.2 and the $-O4$ optimization flag. Because the memory reuse distance data is characterized by a more complex behavior than the dynamic instruction count which we modeled in the previous subsection, we aimed at characterizing the memory access behavior with one variable models. As with the first attempt at constructing one variable models of the FLOP count, we restricted MM to a fixed linear transformation of $NLON$, using $NLON$ as the parameterizing factor. Again we considered the set of problem cases corresponding to a triangular truncation of the spectral coefficients, with MM taking the largest possible value for a given $NLON$: $\lfloor \frac{NLON-1}{3} \rfloor$.

Using our data collection infrastructure, we profiled the SPARC binary to collect the memory reuse distance seen by each load and store instruction. We collected data from multiple executions with different values of the $NLON$ parameter in the range 16 to 288. Then, we applied the modeling strategy described in section 3.4.2 to construct reuse distance models for the memory references in the four most important routines of PSTSWM. We evaluated the resulting models for five different values of $NLON$, and compared the predictions of L1 miss counts and dynamic memory operation counts against measurements with hardware performance counters (HPC) on a MIPS R10000 processor.

Figure 4.4 presents the predicted L1 miss counts for the four routines and five problem sizes considered. The bar graphs are normalized by dividing the predicted values by the corresponding measured values. The $y = 100\%$ line corresponds to the measured counts, or in other words, to an 100% accuracy of the prediction. The prediction is very accurate for the dzp routine. For the other three routines the relative error ranges from 10% to 30%.

Figure 4.5 presents the predicted dynamic memory operation counts for the same four routines and five problem sizes considered before. The counts are shown in rapport to the values measured using the HPC. We notice that the predicted counts are from 2 to 2.7 times larger than the values measured on a MIPS R10000.

The differences between the predicted and the measured values are larger than

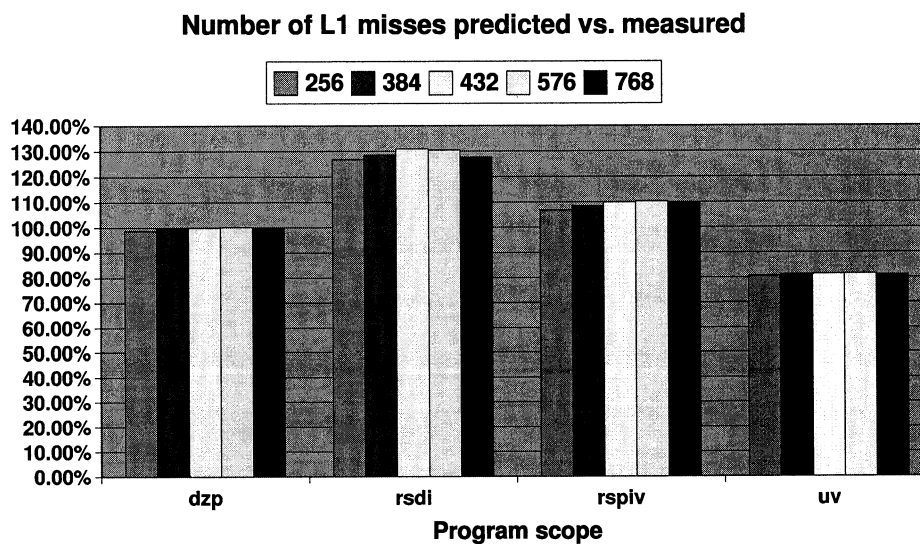


Figure 4.4 : L1 miss counts predicted vs. measured for the four most important computational routines in PSTSWM.

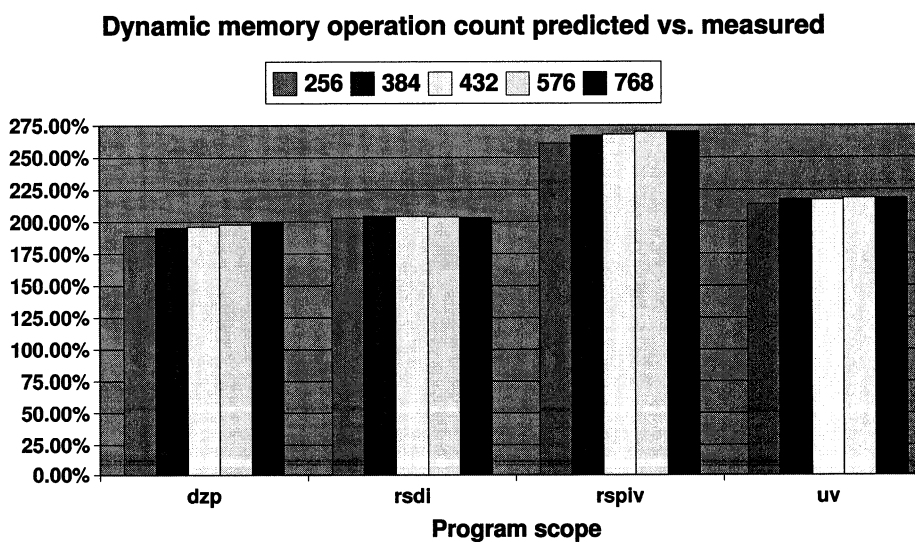


Figure 4.5 : Dynamic memory operation count predicted vs. measured.

what we expected, especially in the case of the dynamic memory operation count. Figure 4.6 presents the innermost loop of routine *rspiv*, loop that accounts for about 90% of all the memory operations in that routine. We notice that accesses to array DATA do not depend on the loop index, therefore DATA elements are temporally reused. Arrays SUM, ALP, and DALP are accessed with stride one. Another observation is that SUM and DATA are arrays of COMPLEX elements, while ALP and DALP are arrays of REAL elements.

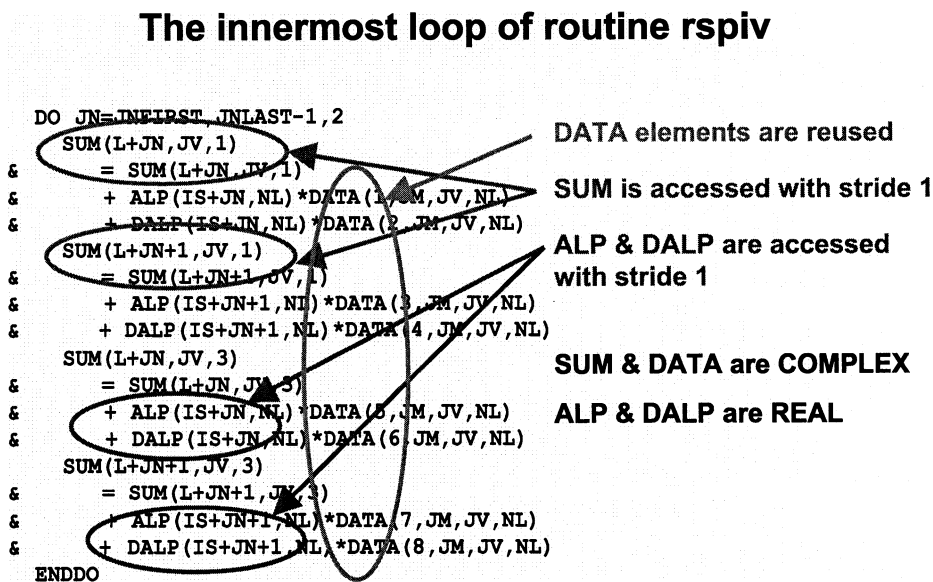


Figure 4.6 : The innermost loop of routine *rspiv*. DATA elements are reused. Arrays SUM, ALP and DALP are accessed with stride 1.

Looking at the assembly code generated by the two compilers, we noticed that the SGI compiler generated 12 loads and 8 stores for the code in the innermost loop of routine *rspiv*, and the Sun compiler generated 33 loads and 21 stores for the same code. By default, Sun compiler aligns the data in COMMON blocks at 4 bytes and generates single word memory instructions to access double- or quad- precision data, even if the highest level of optimization is used. The fortran compiler from Sun requires explicit use of the *-dalign* flag to lay out the double- and quad-precision

data in memory along its natural alignment, and to generate double word memory instructions when accessing this data. This observation explains why the predictions for the number of dynamic memory operations were larger than the measurements on a MIPS R10000 by at least a factor of two. On SPARC, the loop in Figure 4.6 was also unrolled four times, resulting in increased register pressure. Thus, some of the DATA elements could not be reused in registers, resulting in an even larger number of loads and stores being generated.

However, these observations do not explain the inaccuracy of the predictions in the case of the L1 miss counts. All memory accesses in the innermost loop of routine *rs piv*, either reference data that was touched the previous iteration, or access the memory with a stride one. Only the first stride one access to a memory block experiences a cache miss. Therefore, it does not matter if four double word loads or eight single word loads are used to access the content of a 32 bytes memory block, only the first load sees a large reuse distance. The number of misses should be equal to the number of memory blocks accessed in the inner loop. Also, the spill code generated by the Sun compiler has temporal reuse and does not create extra misses. Another factor causes the discrepancies between the predicted and measured L1 miss counts.

The SUM array has COMPLEX elements. Therefore each element of the SUM array occupies 16 bytes. Two COMPLEX elements are packed in a 32 bytes cache line, and two elements from the same column of the SUM array are accessed in one iteration. Depending on the alignment of each column (see Figure 4.7), the first load corresponding to either the first or the second SUM element experiences a large reuse distance, and the other three double word or seven single word loads should see a small reuse distance.

Depending on the problem size, the SUM array has either an even or an odd number of elements on each column. The alignment of SUM's columns alternates when there are an odd number of elements on each column. While the overall number of misses in one loop iteration does not depend on columns' alignment, the reuse

Alignment of SUM array's columns

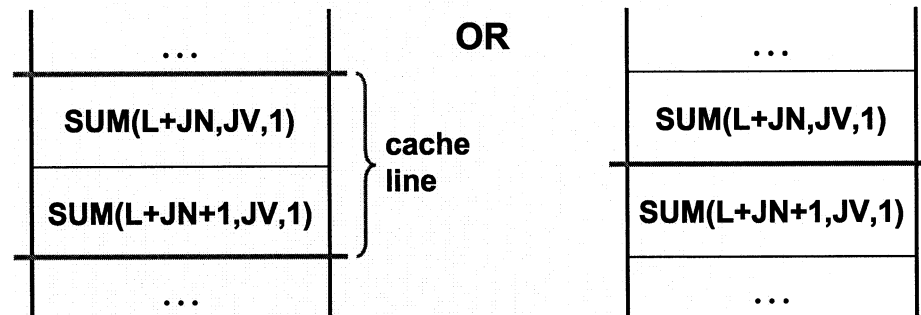


Figure 4.7 : Depending on the alignment of each column of the SUM array, the two elements referenced in one iteration can be located either in the same memory block(left), or in different memory blocks(right).

distance data collected for individual memory instructions is affected. The two load instructions that have the role of first accesses to a memory block in the two presented scenarios, share the number of misses with different ratios for different problem sizes. As a result, the reuse distance models for the two instructions are imprecise. The ALP and DALP arrays, which are also accessed with a stride one, suffer from the same alignment problems.

Figure 4.8 presents the data collected for one of the accesses to the SUM array. We can notice that the reuse distance data corresponding to about half of all the problem sizes for which we collected data, have an uniform ratio between the number of accesses with distance zero and the number of accesses that experienced a large reuse distance (about 10% of the accesses have distance zero). For the other problem sizes, the fraction of accesses with distance zero varies between 50% and 90%. We decided to remove the data corresponding to these divergent problem sizes and to rebuild the reuse distance models for the four routines considered.

Figure 4.9 presents the predictions of L1 miss counts after the divergent data was removed. The accuracy of the predictions improved for each routine and is consistent across the five problem sizes considered.

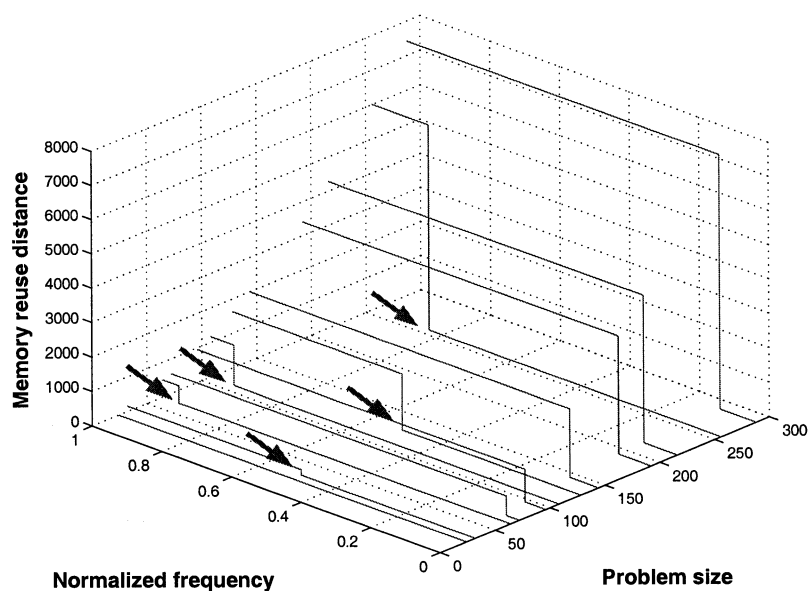


Figure 4.8 : Data collected for one of the accesses to the SUM array. About half of the reuse distance histograms have a similar fraction of accesses with distance zero (10%). The other histograms, marked with an arrow, seem to have a variable fraction of accesses with distance zero.

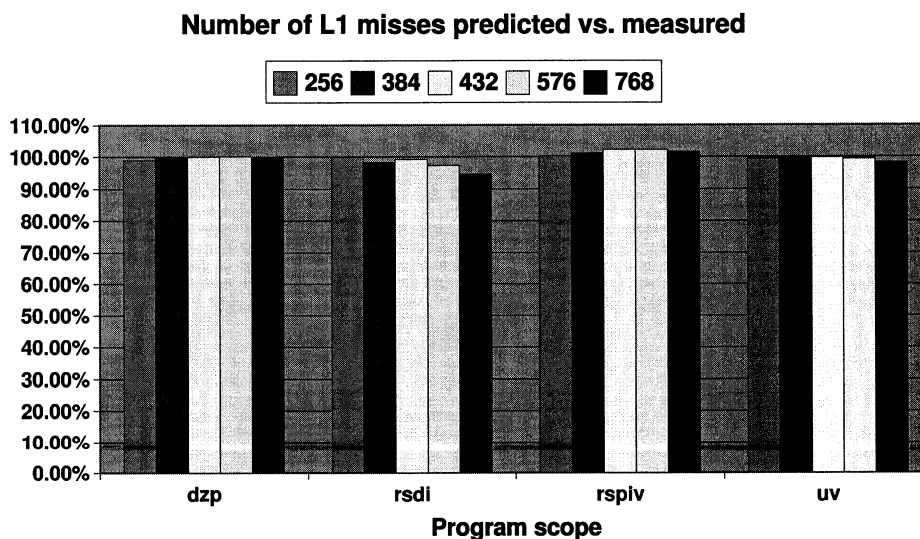


Figure 4.9 : L1 miss count predicted vs. measured after the divergent data points have been removed.

We recompiled the SPARC binary using the `-dalign` flag and generated new predictions for the number of dynamic memory operations. The new predictions are presented in Figure 4.10.

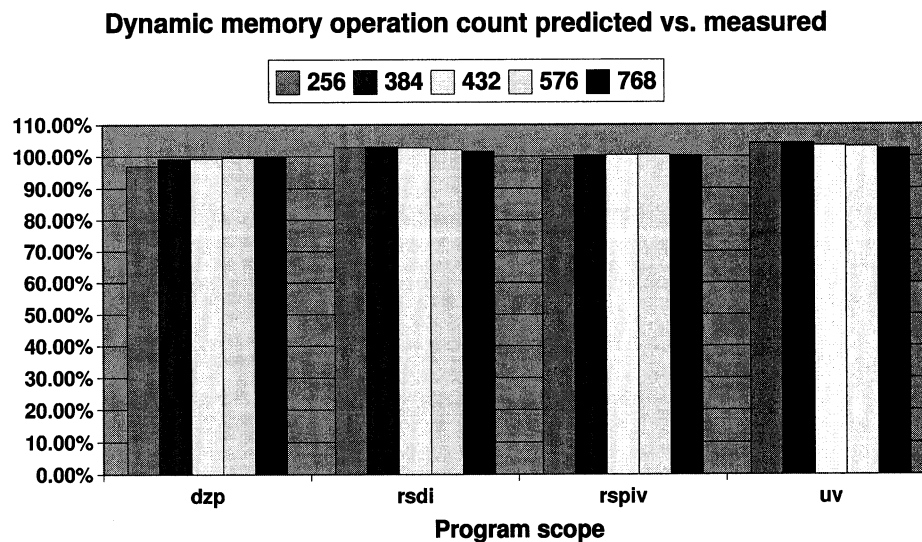


Figure 4.10 : Dynamic memory operation count predicted vs. measured for the binary compiled with `-dalign`.

This experiment uncovers some limitations of the tool. The current approach can model temporal reuse distance accurately. Modeling the spatial reuse distance may be problematic. Unlike temporal reuse which is a pure application characteristic, spatial reuse is sensitive to data alignment and array dimensions, and depends on the cache line size.

4.2 FLOP Count Models for CRM

The Column Radiation Model (CRM) application is a standalone version of the column radiation code employed by the NCAR Community Climate Model (CCM3).

The CRM application has two input parameters that can be independently modified and that affect the number of floating point instructions executed. The two input

parameters, LON and LAT , can be any natural number without other imposed constraints. Using a process similar to the one described in Section 4.1.1, we performed a series of basic block execution frequency measurements, for different values of the input parameters LON and LAT .

Using the filter tool on the collected data files, we extracted instruction count information for each floating point class of instructions. Besides multiplies and additions, the CRM executes also a significant number of division and square root operations. For the CRM application we built separate models for each loop and routine scope in the program. This detailed modeling process produced more than 700 two variable models. This thesis presents only the models of the FLOP counts aggregated at the entire program level.

First, we built models parameterized by LAT for a fixed value of LON . We repeated this process for different values of LON , obtaining an equal number of models in LAT for each floating point instruction type. Next, we derived models parameterized by LON for the coefficients of the models in LAT (see Section 4.1.1). The resulting two parameter models perfectly approximate the measured counts for all the instruction types:

$$FpAdd = (145718 * LON + 1906) * LAT + 54 * LON + 599 \quad (4.13)$$

$$FpMult = (210726 * LON + 1964) * LAT + 39 * LON + 1057 \quad (4.14)$$

$$FpDiv64 = (41172 * LON + 93) * LAT + 37 * LON + 277 \quad (4.15)$$

$$FpSqrt64 = (15451 * LON + 1) * LAT + 2 \quad (4.16)$$

The FLOP count models for the CRM are a complete success. We were able to automatically synthesize accurate two parameter models of the floating point instruction count for every loop in the program.

Routine	Error	Derived models
cftb	0.0096	$FpAdd = 1057.14 * NLON^2 \log_2(NLON) - 1723.15 * NLON^2 + 20425.5 * NLON - 244116$
	0.0394	$FpMult = 295.77 * NLON^2 \log_2(NLON) + 1035.57 * NLON^2 - 40792.15 * NLON + 487761$
cftf	0.011	$FpAdd = 1200.58 * NLON^2 \log_2(NLON) - 135.06 * NLON^2 - 32666.38 * NLON + 390496$
	0.0394	$FpMult = 473.23 * NLON^2 \log_2(NLON) + 1656.91 * NLON^2 - 65267.44 * NLON + 780417$
dpupim2	0	$FpAdd = 602.5 * MM^2 + 1807.5 * MM + 1205$
	0	$FpMult = 723 * MM^2 + 2169 * MM + 2651$
dzp	0	$FpAdd = 180.75 * NLON * MM^2 + 1265.25 * NLON * MM + 1084.5 * NLON$
	0	$FpMult = 180.75 * NLON * MM^2 + 542.25 * NLON * MM + 361.5 * NLON$
extract	0	$FpAdd = 2410 * NLON^2 + 1928 * NLON$
	0	$FpMult = 1928 * NLON^2 + 1928 * NLON + 241$
indi	0	$FpAdd = 482 * MM^2 + 1446 * MM + 964$
	0	$FpMult = 723 * MM^2 + 2169 * MM + 1446$
inject	0	$FpAdd = 1506.25 * NLON^2 + 1205 * NLON$
	0	$FpMult = 602.5 * NLON^2$
inpiv	0	$FpAdd = 482 * MM^2 + 1446 * MM + 964$
	0	$FpMult = 964 * MM^2 + 2892 * MM + 1928$
nonlim	0	$FpAdd = 361.5 * NLON^2$
	0	$FpMult = 1084.5 * NLON^2 + 120.5 * NLON + 723$
rsdi	0	$FpAdd = (180.75 * NLON - 723) * MM^2 + (542.25 * NLON - 2169) * MM + (361.5 * NLON - 1446)$
	0	$FpMult = (180.75 * NLON - 723) * MM^2 + (542.25 * NLON - 2169) * MM + (361.5 * NLON - 1446)$
rspiv	0	$FpAdd = (241 * NLON - 964) * MM^2 + (723 * NLON - 2892) * MM + (482 * NLON - 1928)$
	0	$FpMult = (241 * NLON - 964) * MM^2 + (723 * NLON - 2892) * MM + (482 * NLON - 1928)$
simplic	0	$FpAdd = 723$
step	0	$FpMult = 240$

Table 4.1 : Two variable routine-level models of the count of dynamic floating point instructions and their relative error (first part).

Routine	Error	Derived models
tmpdi	0	$FpAdd = 1446 * NLON * MM + 1446 * NLON$
	0	$FpMult = 1265.25 * NLON * MM + 1566.5 * NLON$
tmppiv	0	$FpAdd = 2410 * NLON * MM + 2410 * NLON$
	0	$FpMult = 2048.5 * NLON * MM + 2470.25 * NLON$
tmpuv	0	$FpAdd = 482 * MM^2 + 1446 * MM + 2410$
	0	$FpMult = 1446 * MM^2 + 4338 * MM + 3374$
uv	0	$FpAdd = 241 * NLON * MM^2 + 1205 * NLON * MM + 964 * NLON$
	0	$FpMult = 241 * NLON * MM^2 + 723 * NLON * MM + 482 * NLON$

Table 4.2 : Two variable routine-level models of the count of dynamic floating point instructions and their relative error (second part).

Chapter 5

Conclusions

This thesis proposes, implements and demonstrates the viability of a new performance prediction method, which analyzes and models application characteristics independently of the target architecture. A binary rewriting tool instruments an application's binary to collect data about the most important application-related factors that affect run-time performance: the execution frequency of each basic block in the program, the memory access pattern exhibited by each memory reference, and the volume and frequency of communication among processes. A post-processing tool processes the collected data into accurate architecture-neutral models. The models, parameterized by problem size, enable us to predict important characteristics of the application without the need to execute the program at scale. An instantiation of the models for a set of parameters and a description of a target architecture enable us to predict the application's performance for an arbitrary machine.

The characteristics of regular scientific programs can be modeled with a high level of accuracy using the proposed approach. In our experiment modeling the memory reuse distance for the PSTSWM application, we encountered difficulties in building an accurate model of distance because of anomalies introduced by spatial reuse. Unlike temporal reuse distance which is an application specific factor, spatial reuse distance seen by individual references is affected by the layout of data in memory and by the size of the cache line. More study is needed to determine if a different approach for accounting for the impact of spatial reuse can offset the effects of data alignment on the measured spatial reuse distance and increase the confidence in the accuracy of the cache miss predictions. A possible but untested solution is to aggregate the data

from the instructions that belong to the same loop and are characterized by circular reuse on each other's accessed data.

In addition to supporting accurate cross-architecture predictions, our configurable scheduler enables us to determine the parts of a program that run most inefficiently. Moreover, it enables us to understand what are the causes of inefficiency: limited memory bandwidth, schedule dependencies among instructions on the critical path, or a mix of instructions that is not balanced with respect to the number and type of functional units on the target architecture. To facilitate the use of this performance data, we plan to extend the HPCView [24] performance data analysis tool to present the kinds of information produced by our tool. HPCView can correlate our performance data to the source code and its user interface will enable us to browse the performance data in a top-down way that will facilitate its interpretation.

In the future, we plan to translate our data on memory reuse distance into a prediction of latency for a target memory hierarchy. With this data in hand, we plan extending the scheduler to consider the individual latency experienced by each memory reference. For the long term, more research is necessary to investigate if the approach presented in this thesis can be extended to parallel applications by considering the effects of synchronization and serialization on an application's performance. One of the major difficulties in characterizing the performance of parallel applications is to automatically understand the communication pattern exhibited by the application and to represent this pattern in a form that enables scalability predictions.

Bibliography

- [1] The ASCI Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative.
http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.
- [2] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dubeb, E. N. Houstis, J. R. Rice, R. Sakellariou, D. Sundaram-Stukel, P. T. Teller, and M. K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *Software Engineering*, 26(11):1027–1048, 2000.
- [3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
- [4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.
- [5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
- [7] B. Bennett and V. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.
- [8] K. Beyls and E. D’Hollander. Reuse distance as a metric for cache behavior. In

- IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, pages 617–662, 2001.
- [9] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
- [11] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
- [12] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, 2001.
- [13] M. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Principles Practice of Parallel Programming*, pages 276–287, 1997.
- [14] S. Ghosh and M. M. amd Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.
- [15] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX*, 1992.
- [16] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing*, 2000.
- [17] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13(3):313–322, 1973.

- [18] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.
- [19] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, Oct. 1994.
- [20] MathWorks. *Optimization Toolbox: Function quadprog*.
<http://www.mathworks.com/access/helpdesk/help/toolbox/optim/quadprog.shtml>.
- [21] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [22] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarrla-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced Optimization Strategies in the Rice dHPF Compiler. *Concurrency: Practice and Experience*, 2001.
- [23] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the International Conference on Supercomputing (ICS1001)*, pages 154–165, Sorrento, Italy, June 2001.
- [24] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.
- [25] F. Mueller, T. Mohan, B. de R. Supinski, S. A. McKee, and A. Yoo. Partial data traces: Efficient generation and representation. In *Workshop on Binary Translation*, IEEE Technical Committee on Computer Architecture Newsletter, Oct. 2001.
- [26] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. H. Levy, and B. Bershad. Instrumentation and optimization of win32/intel executables using etch.

In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, Seattle, WA, USA, Aug. 1997.

- [27] A. Snaveley, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles, 2001.
- [28] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, May 1994.
- [29] D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*, Atlanta, May 1999.
- [30] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.